UNIT IV: Visible Surface Detection & GPU Architecture

Visible-surface detection

- It is also referred to as hidden-surface elimination methods.
- In a given set of 3D objects and viewing specification, we wish to determine which lines or surfaces of the objects are visible, so that we can display only the visible lines or surfaces. This process is known as hidden surfaces or hidden line elimination, or visible surface determination.
- The hidden line or hidden surface algorithm determines the lines, edges, surfaces or volumes that are visible or invisible to an observer located at a specific point in space.

- We can broadly classify visible-surface detection algorithms according to whether they deal with the object definitions or with their projected images.

- These two approaches are called **object-space** methods and **image-space** methods, respectively:
  - An object-space method compares objects and parts of objects to each other within the scene definition to determine which surfaces, as a whole, we should label as visible.
  - In an image-space algorithm, visibility is decided point by point at each pixel position on the projection plane.

Back-face detection

- A fast and simple object-space method for locating the **back faces** of a polyhedron is based on front-back tests. A point (x, y, z) is behind a polygon surface if

$$Ax + By + Cz + D < 0 \qquad \textbf{(1)}$$

  where A, B, C, and D are the plane parameters for the polygon.
- When this position is along the line of sight to the surface, we must be looking at the back of the polygon. Therefore, we could use the viewing position to test for back faces.

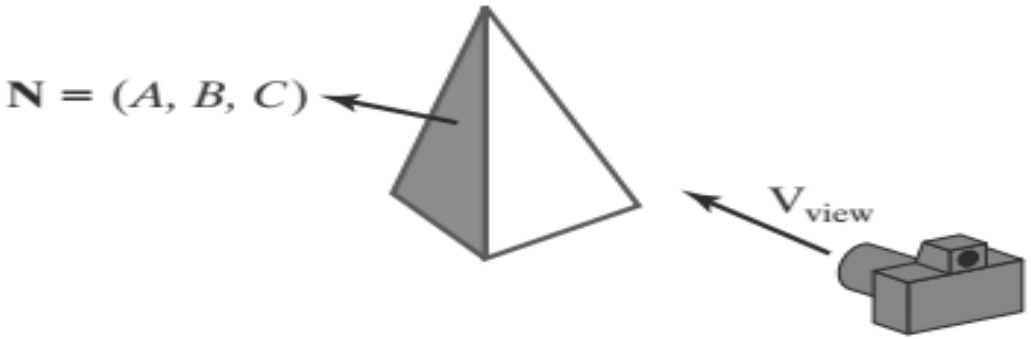


FIGURE 1
A surface normal vector **N** and the viewing-direction vector $\mathbf{V}_{view}$.

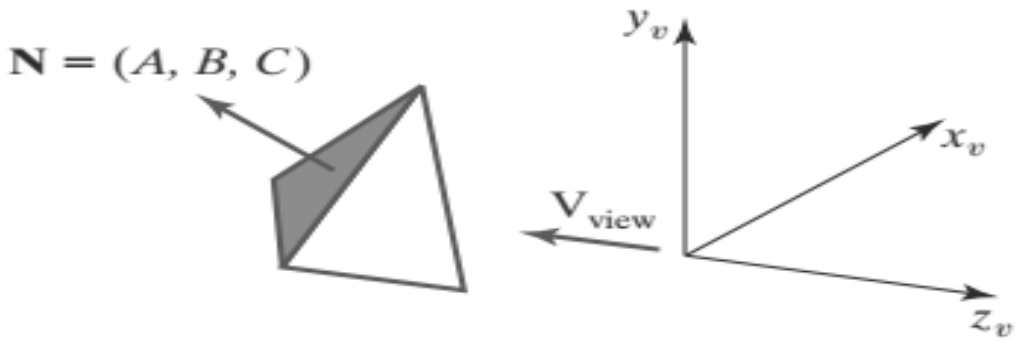


FIGURE 2
A polygon surface with plane parameter $C < 0$ in a right-handed viewing coordinate system is identified as a back face when the viewing direction is along the negative z_v axis.

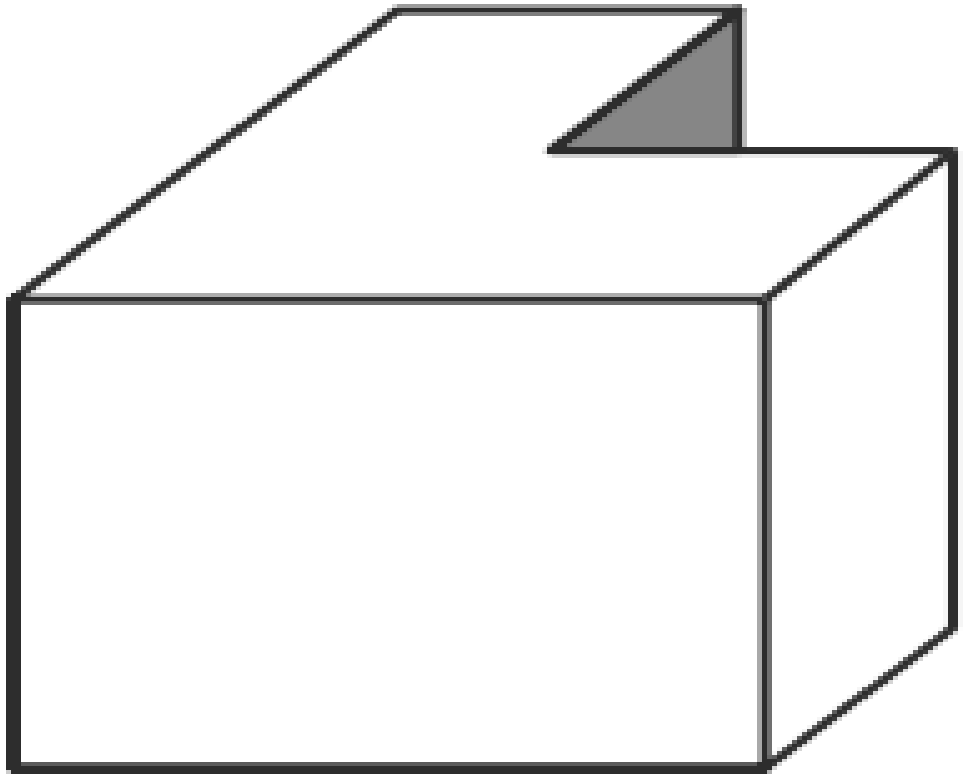


FIGURE 3
View of a concave polyhedron with one face partially hidden by other faces of the object.

Depth-buffer Method

- A commonly used image-space approach for detecting visible surfaces is the **depth-buffer method,** which compares surface depth values throughout a scene for each pixel position on the projection plane.
- Each surface of a scene is processed separately, one pixel position at a time, across the surface.
- The algorithm is usually applied to scenes containing only polygon surfaces, because depth values can be computed very quickly and the method is easy to implement.
- This visibility-detection approach is also frequently alluded to as the *z-buffer method,* because object depth is usually measured along the *z* axis of a viewing system.

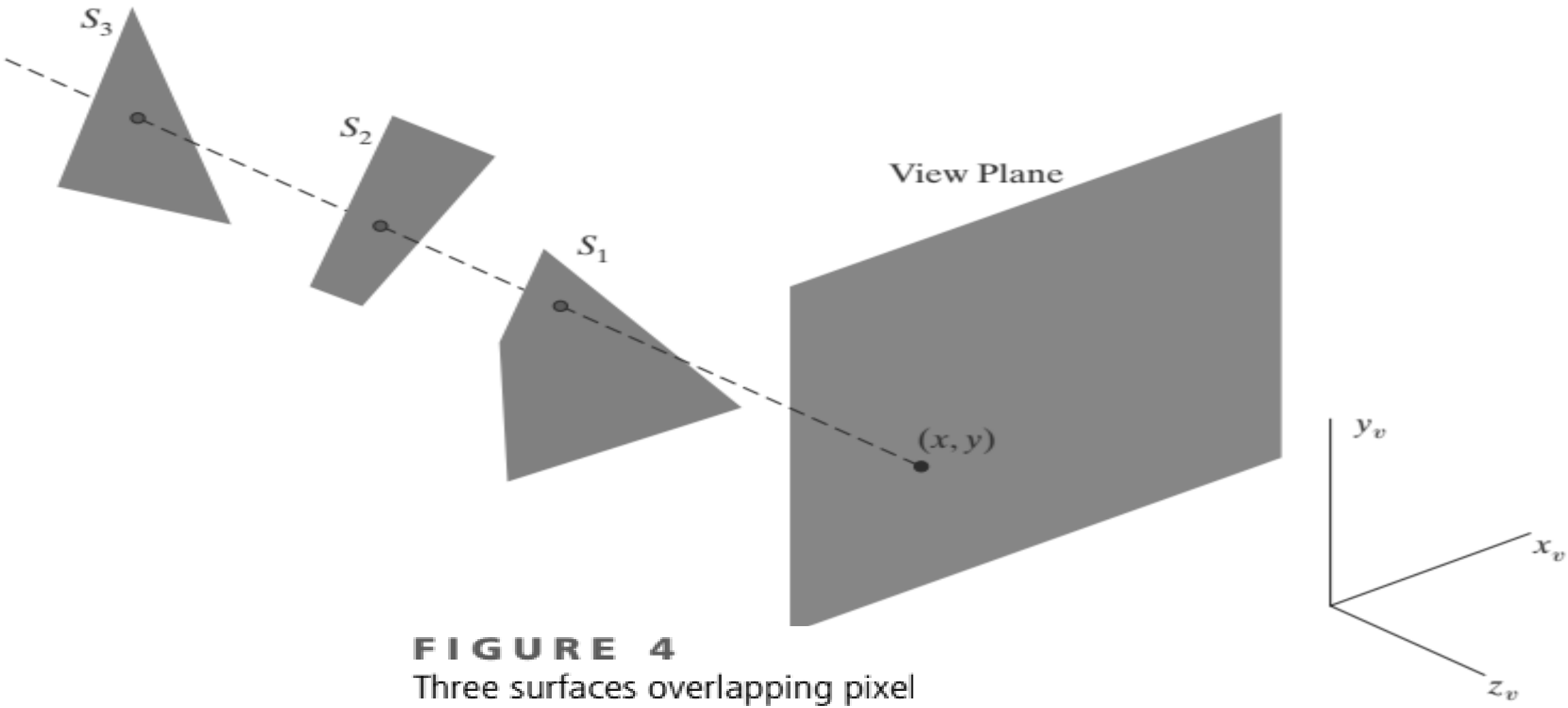


FIGURE 4
Three surfaces overlapping pixel position (x, y) on the view plane. The visible surface, S_1, has the smallest depth value.

Depth-Buffer Algorithm

1. Initialize the depth buffer and frame buffer so that for all buffer positions (x, y),

```
depthBuff (x, y) = 1.0,    frameBuff (x, y) = backgndColor
```

2. Process each polygon in a scene, one at a time, as follows:
   - For each projected (x, y) pixel position of a polygon, calculate the depth z (if not already known).
   - If $z <$ `depthBuff (x, y)`, compute the surface color at that position and set

```
depthBuff (x, y) = z,    frameBuff (x, y) = surfColor (x, y)
```

After all surfaces have been processed, the depth buffer contains depth values for the visible surfaces and the frame buffer contains the corresponding color values for those surfaces.

At surface position (x, y), the depth is calculated from the plane equation as

$$z = \frac{-Ax - By - D}{C} \tag{4}$$

For any scan line (Figure 5), adjacent horizontal x positions across the line differ by ± 1, and vertical y values on adjacent scan lines differ by ± 1. If the depth of position (x, y) has been determined to be z, then the depth z' of the next position $(x + 1, y)$ along the scan line is obtained from Eq. 4 as

$$z' = \frac{-A(x + 1) - By - D}{C} \tag{5}$$

or

$$z' = z - \frac{A}{C} \tag{6}$$

The ratio $-A/C$ is constant for each surface, so succeeding depth values across a scan line are obtained from preceding values with a single addition.

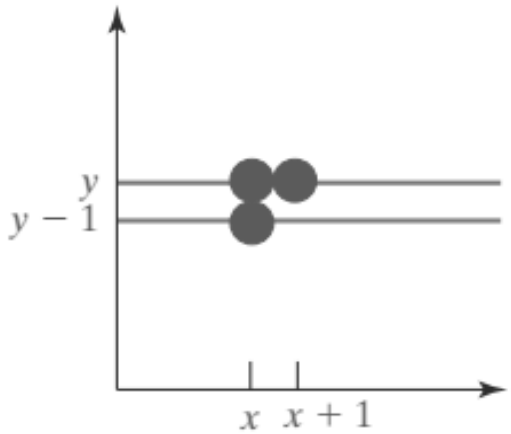


FIGURE 5
From position (x, y) on a scan line, the next position across the line has coordinates $(x + 1, y)$, and the position immediately below on the next line has coordinates $(x, y - 1)$.

- **Drawback:** A drawback of the depth-buffer method is that it identifies only one visible surface at each pixel position. In other words, it deals only with opaque surfaces and cannot accumulate color values for more than one surface, as is necessary if transparent surfaces are to be displayed.

A-Buffer Method

- An extension of the depth-buffer ideas is the **A-buffer** procedure (at the other end of the alphabet from "*z*-buffer," where *z* represents depth).
- This depth-buffer extension is an antialiasing, area-averaging, visibility-detection method developed at Lucasfilm Studios for inclusion in the surface-rendering system called REYES (an acronym for "Renders Everything You Ever Saw").
- The buffer region for this procedure is referred to as the *accumulation buffer,* because it is used to store a variety of surface data, in addition to depth values.

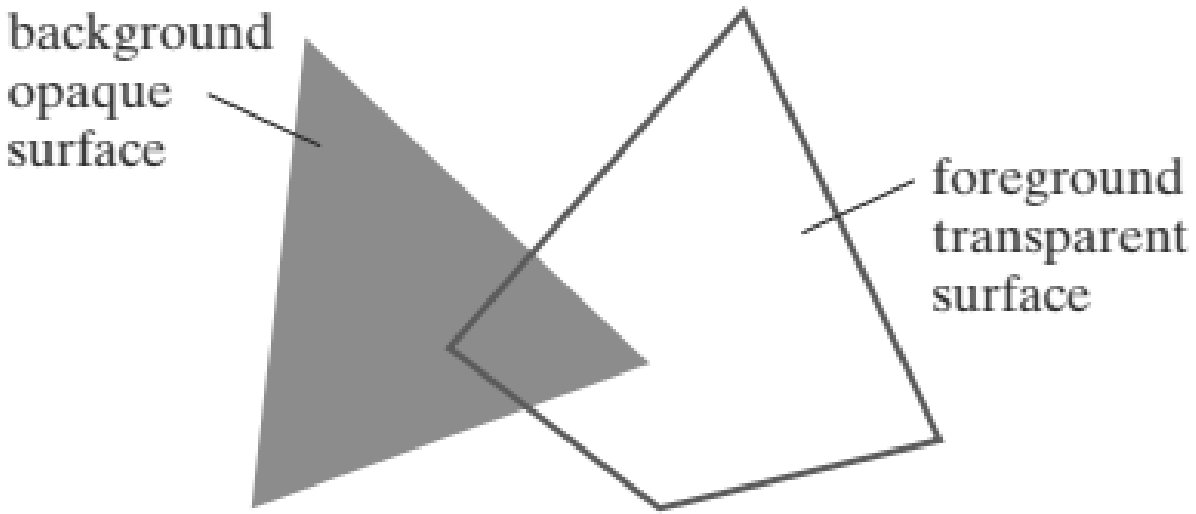


FIGURE 8
Viewing an opaque surface through a transparent surface requires multiple color inputs and the application of color-blending operations.

Each position in the A-buffer has two fields:

- Depth field: Stores a real-number value (positive, negative, or zero).
- Surface data field: Stores surface data or a pointer.

Surface information in the A-buffer includes:

- RGB intensity components
- Opacity parameter (percent of transparency)
- Depth
- Percent of area coverage
- Surface identifier
- Other surface-rendering parameters

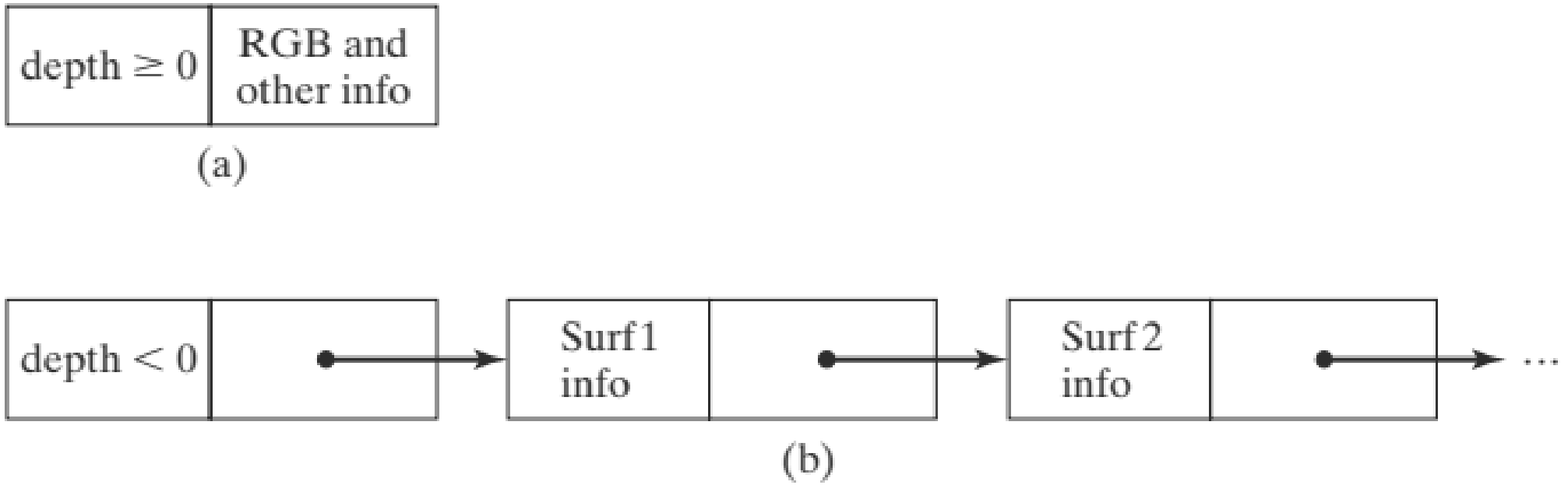


FIGURE 9

Two possible organizations for surface information in an A-buffer representation for a pixel position. When a single surface overlaps the pixel, the surface depth, color, and other information are stored as in (a). When more than one surface overlaps the pixel, a linked list of surface data is stored as in (b).

Scan-line Method

- This image-space method for identifying visible surfaces computes and compares depth values along the various scan lines for a scene.
- As each scan line is processed, all polygon surface projections intersecting that line are examined to determine which are visible.
- Across each scan line, depth calculations are performed to determine which surface is nearest to the view plane at each pixel position.
- When the visible surface has been determined for a pixel, the surface color for that position is entered into the frame buffer.
- Surfaces are processed using the information stored in the polygon tables.

- The edge table contains coordinate endpoints for each line in the scene, the inverse slope of each line, and pointers into the surface-facet table to identify the surfaces bounded by each line.
- The surface-facet table contains the plane coefficients, surface material properties, other surface data, and possibly pointers into the edge table.
- To facilitate the search for surfaces crossing a given scan line, an active list of edges is formed for each scan line as it is processed.
- The active edge list contains only those edges that cross the current scan line, sorted in order of increasing x.
- In addition, we define a flag for each surface that is set to “on” or “off” to indicate whether a position along a scan line is inside or outside the surface.

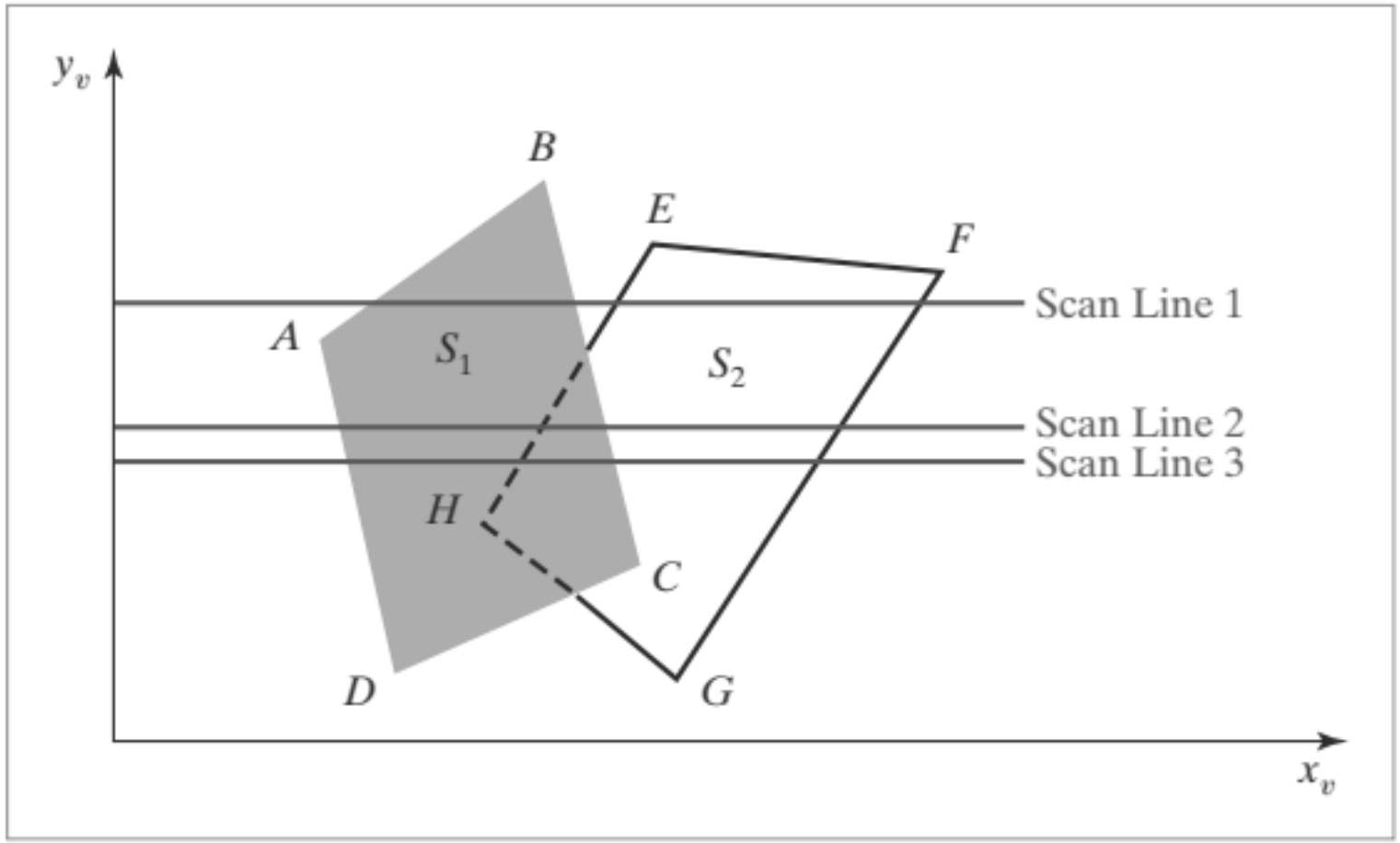


FIGURE 10
Scan lines crossing the view-plane projection of two surfaces, S_1 and S_2. Dashed lines indicate the boundaries of hidden surface sections.

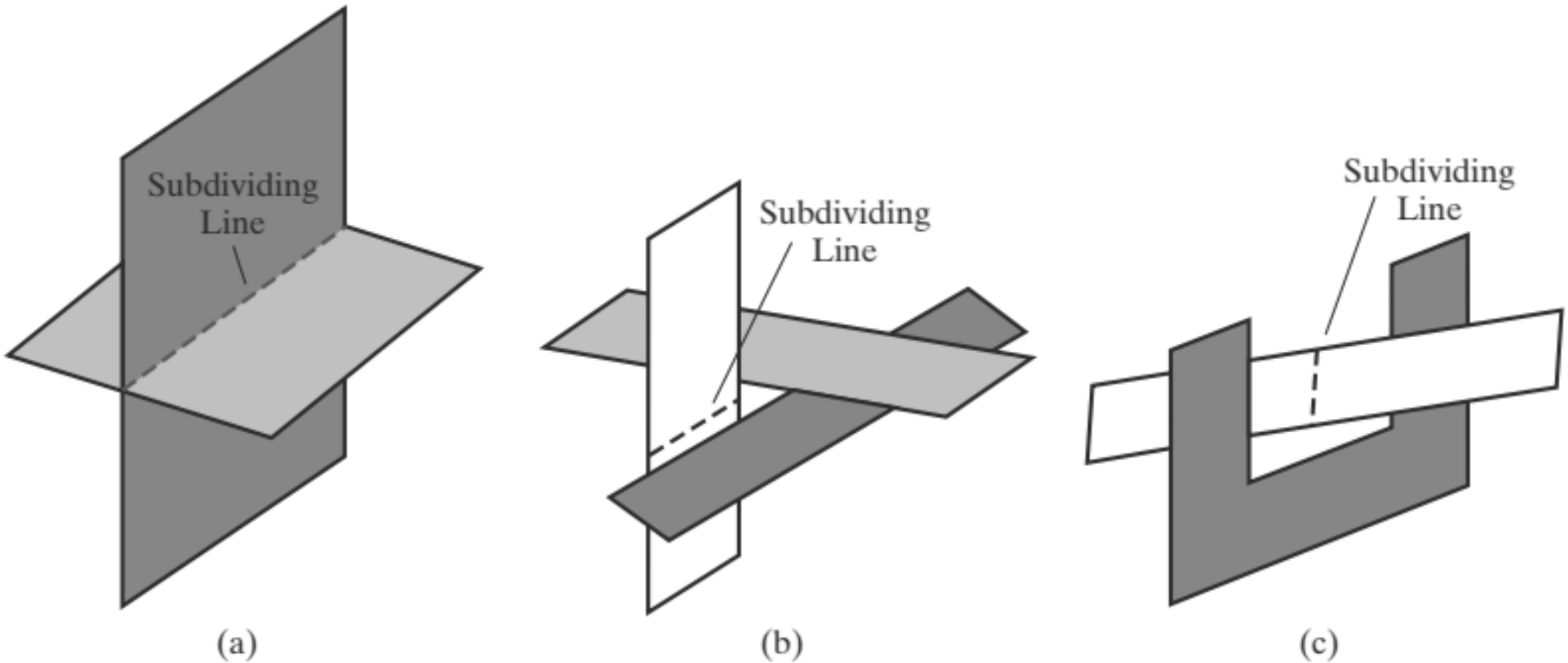


FIGURE 11
Intersecting and cyclically overlapping surfaces that alternately obscure one another.

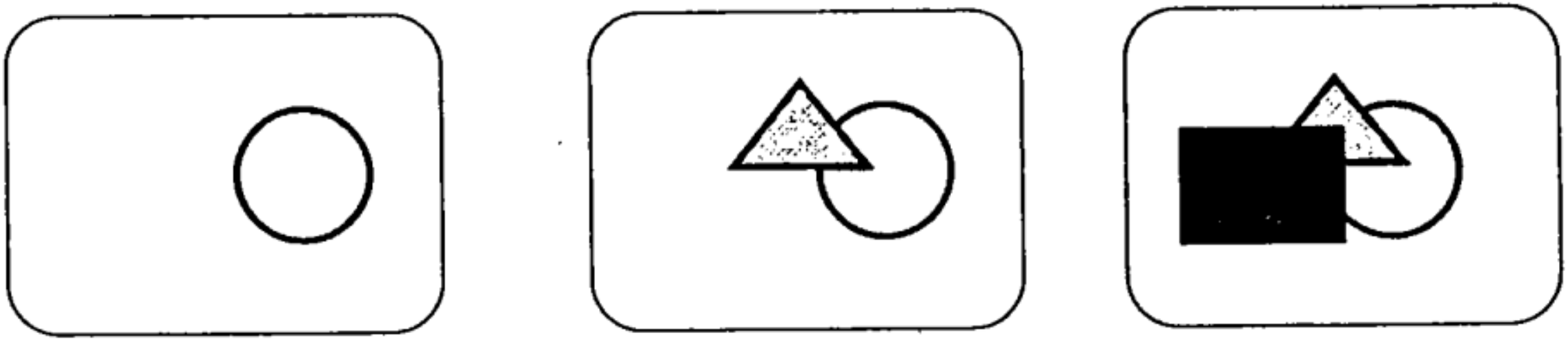
Depth-sorting Method

- Using both image-space and object-space operations, the **depth-sorting** method performs the following basic functions:
  1. Surfaces are sorted in order of decreasing depth.
  2. Surfaces are scan-converted in order, starting with the surface of greatest depth.

- Sorting operations are carried out in both image and object space, and the scan conversion of the polygon surfaces is performed in image space.
- This visibility-detection method is often referred to as the **painter's algorithm.**

Painter's Algorithm

1. The z-extents of P and Q do not overlap, i.e. $z_{Q\,max} < z_{P\,min}$ (see Fig. 8.8 (a))
2. The y-extents of P and Q do not overlap (see Fig. 8.8 (b))
3. The x-extents of P and Q do not overlap
4. Polygon P lying entirely on the opposite side of Q's plane from the view port. (see Fig. 8.8 (c))
5. Polygon Q lying entirely on the same side of P's plane as the viewport. (see Fig. 8.8 (d)).
6. The projections of the polygons P and Q onto the xy screen do not overlap.



1. Sort all polygons in order of decreasing depth.
2. Determine all polygons Q (preceding P) in the polygon list whose z-extents overlap that of P.
3. Perform test 2 through 6 for each Q

   a) If every Q passes the tests, scan convert the polygon P.

   b) If test fails for some Q, swap P and Q in the list, and make the indication that Q is swapped. If Q has already been swapped, use the plane containing polygon P to divide polygon Q into two polygons, Q_1 and Q_2. Replace Q with Q_1 and Q_2. Repeat step 3.

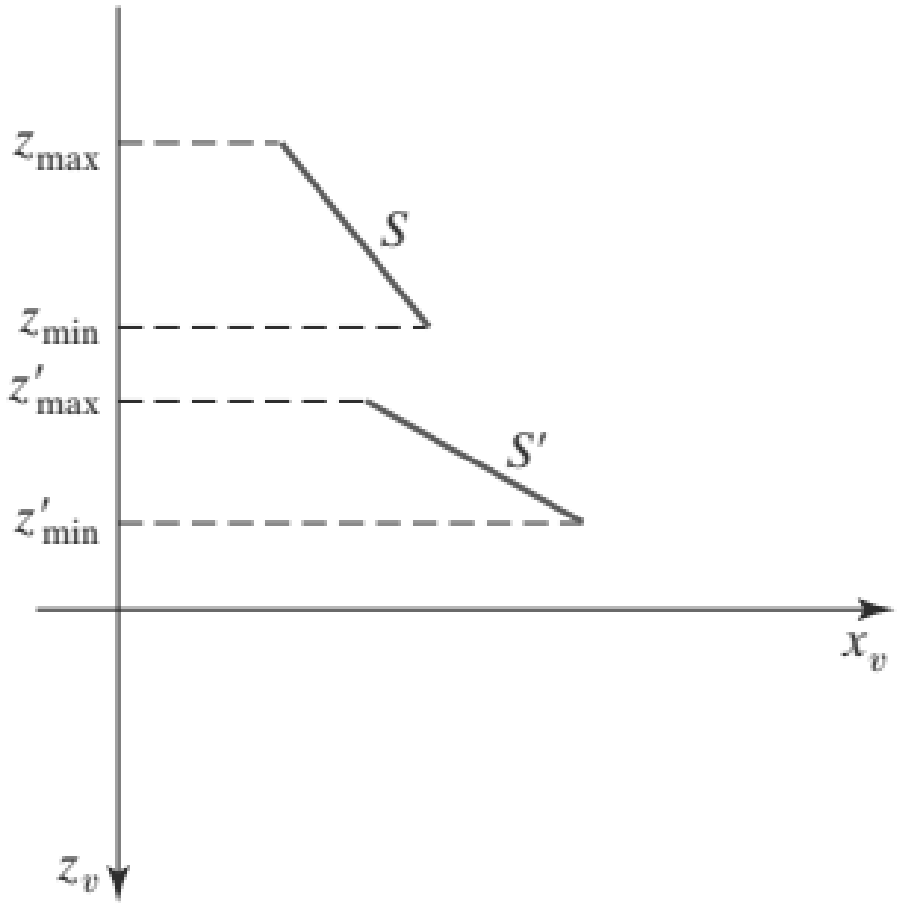


FIGURE 12
Two surfaces with no depth overlap.

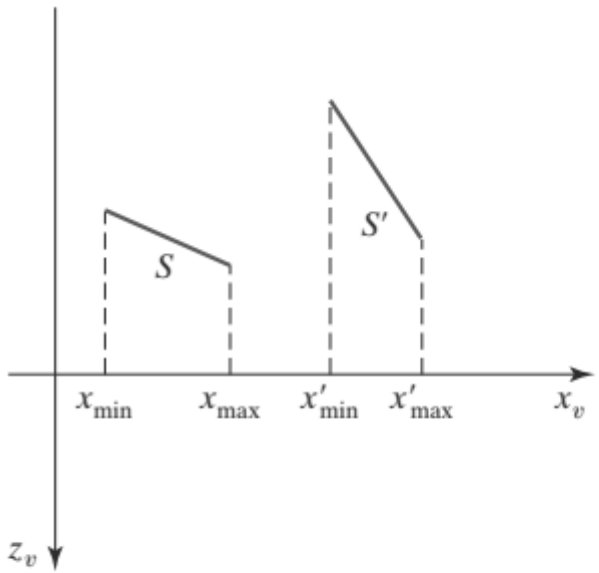


FIGURE 13
Two surfaces with depth overlap but no overlap in the x direction.

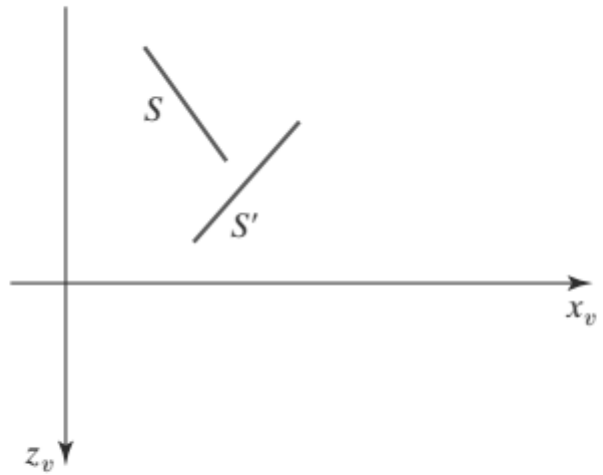


FIGURE 14
Surface S is completely behind the overlapping surface S'.

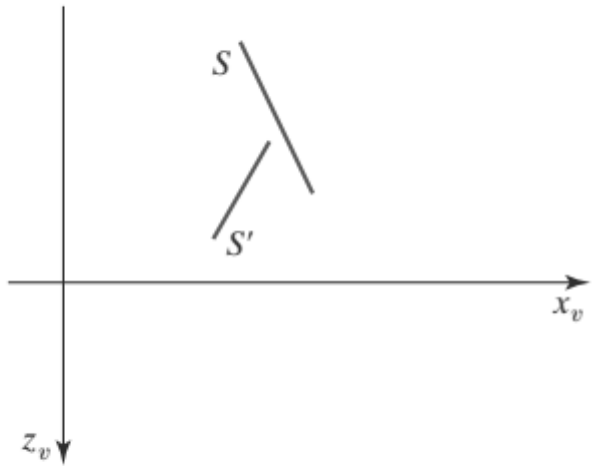


FIGURE 15
Overlapping surface S' is completely in front of surface S, but S is not completely behind S'.

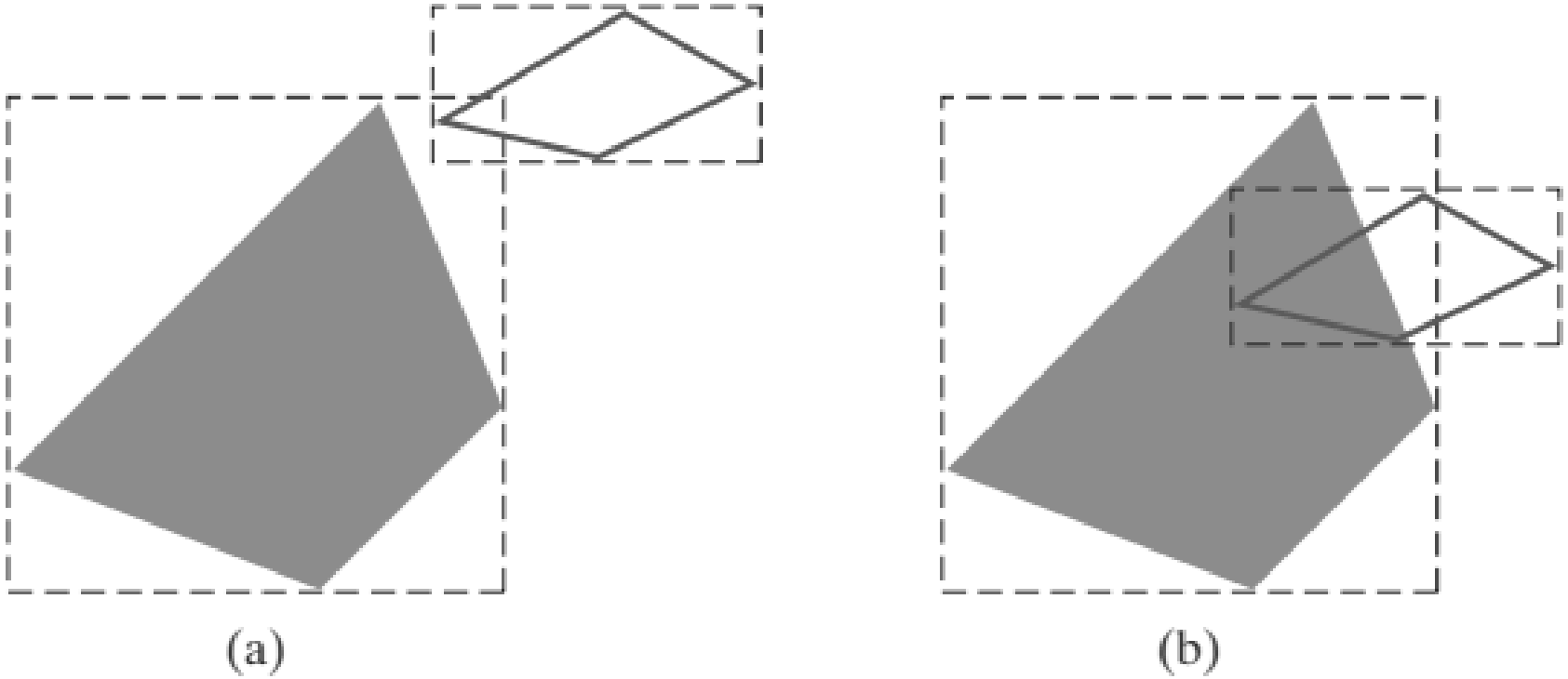


FIGURE 16
Two polygon surfaces with overlapping bounding rectangles in the *x y* plane.

Constructive Solid-Geometry Methods

- This method creates a new volume using Boolean set operations (e.g.- union, intersection or difference operation) on two specified volumes.
- Initially a set of primitive 3d volumes are available e.g.- cubes, spheres ,pyramids, cylinders, cones and closed spline surfaces.

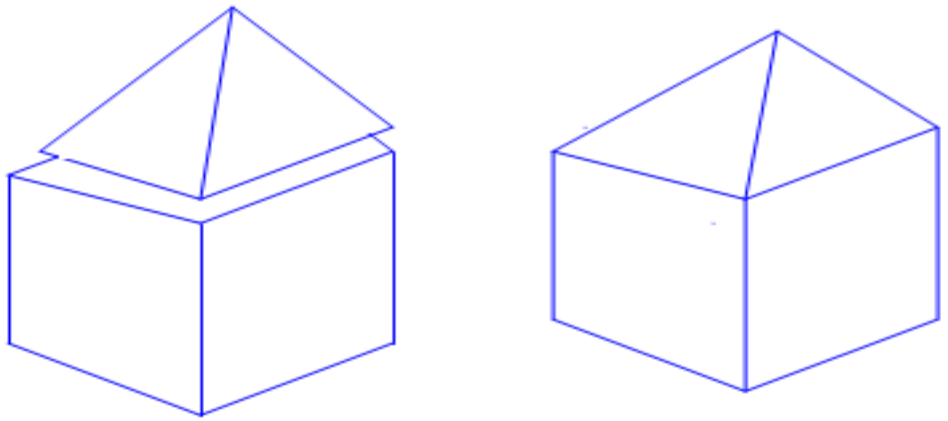
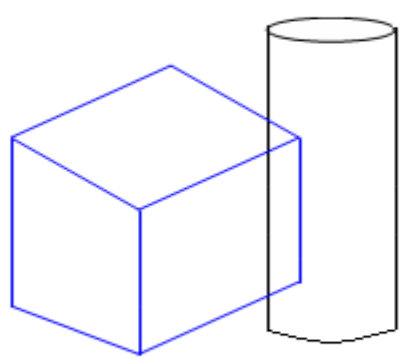
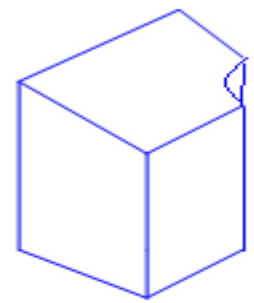


Fig : combining two objects with union operation



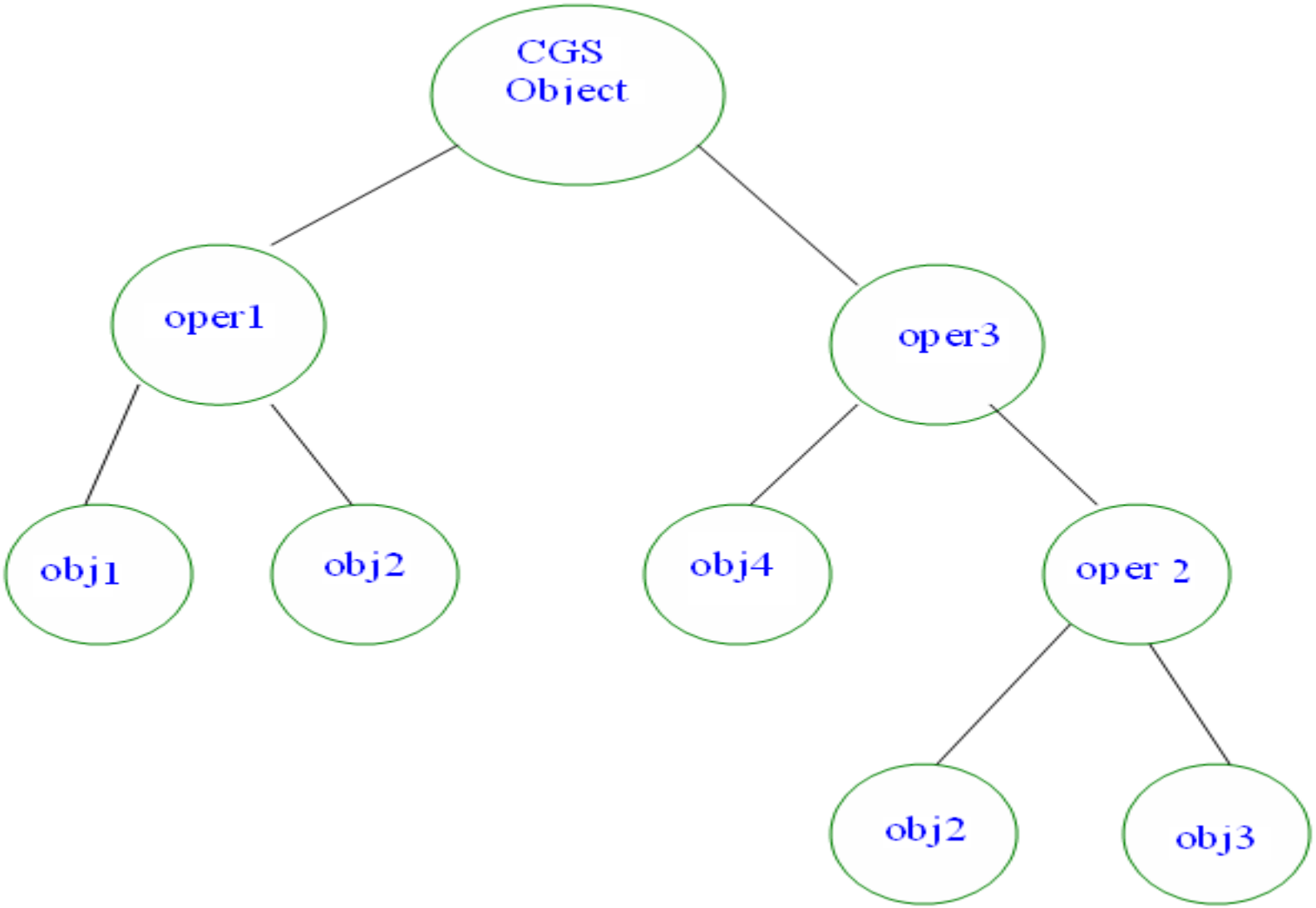
Overlapping objects

Result of intersection operation



Result of difference operation

Tree representation for a CGS object



Binary Space Partitioning (BSP) Tree

•One of class of "list-priority" algorithms – returns ordered list of polygon fragments for specified view point (static pre-processing stage).

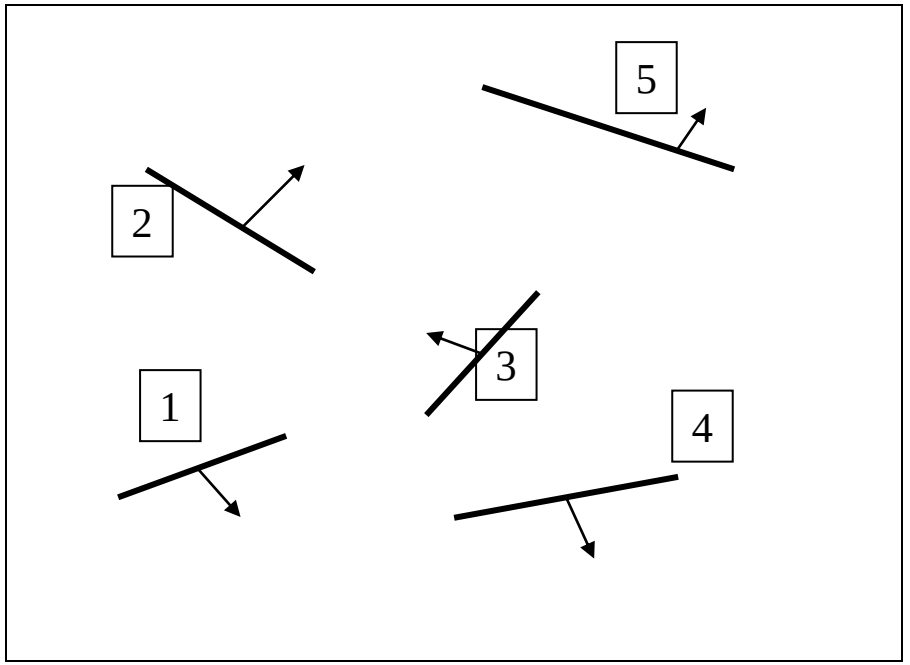
•Choose polygon arbitrarily

•Divide scene into front (relative to normal) and back half-spaces.

•Split any polygon lying on both sides.

•Choose a polygon from each side – split scene again.

•Recursively divide each side until each node contains only 1 polygon.



View of scene from above

BSP Tree

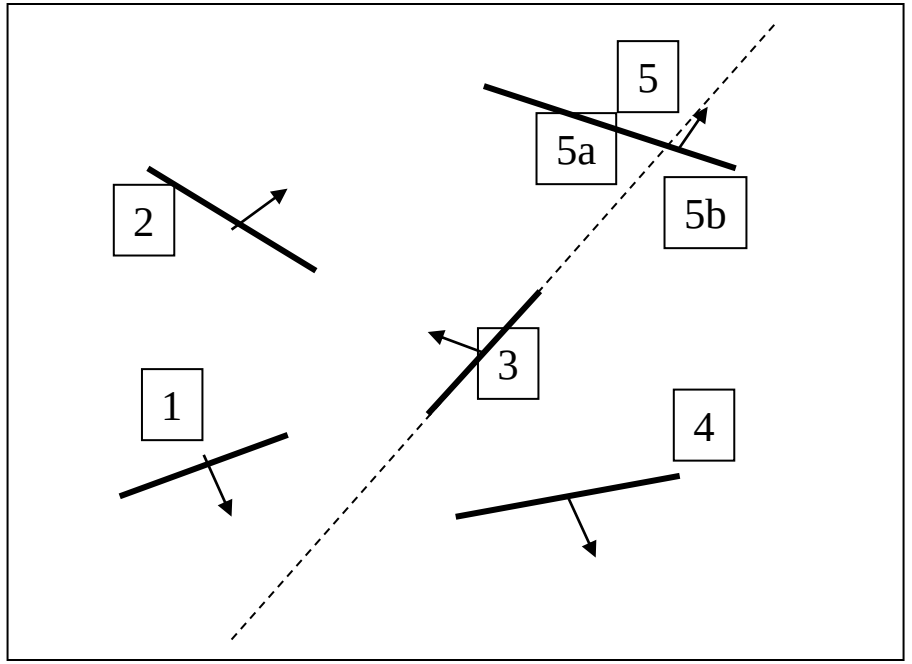
•**Choose polygon arbitrarily**

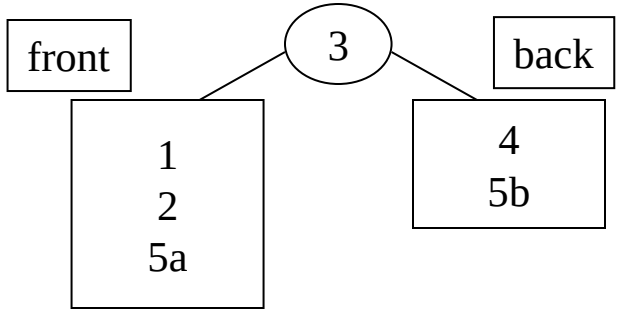
•**Divide scene into front (relative to normal) and back half-spaces.**

•**Split any polygon lying on both sides**.

•Choose a polygon from each side – split scene again.

•Recursively divide each side until each node contains only 1 polygon.





BSP Tree

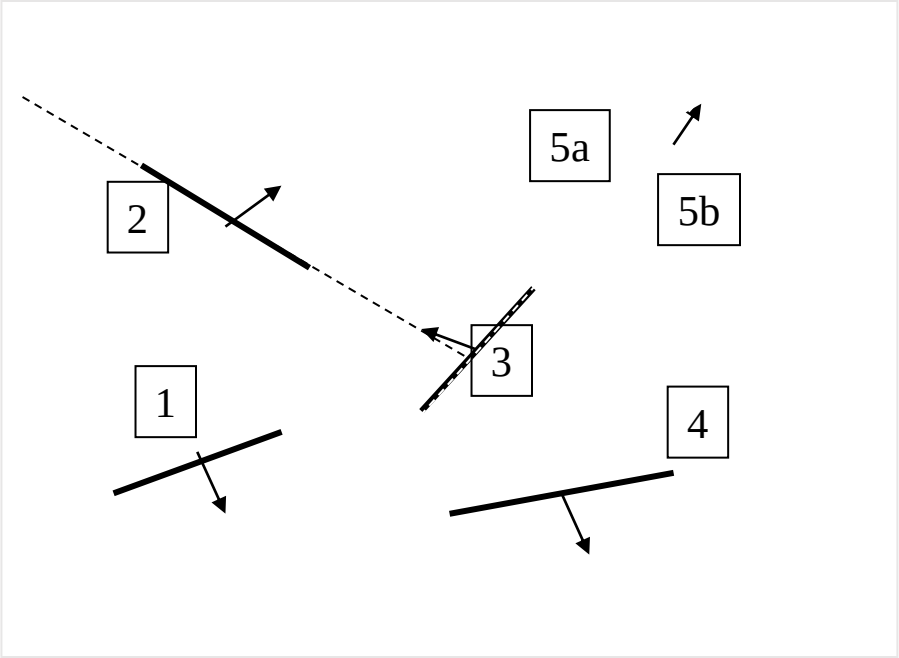
•Choose polygon arbitrarily

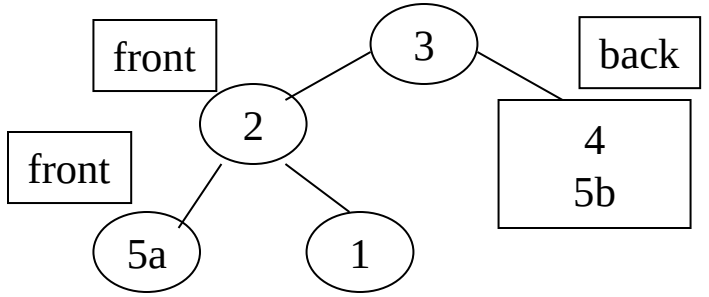
•Divide scene into front (relative to normal) and back half-spaces.

•Split any polygon lying on both sides.

•**Choose a polygon from each side – split scene again.**

•Recursively divide each side until each node contains only 1 polygon.





BSP Tree

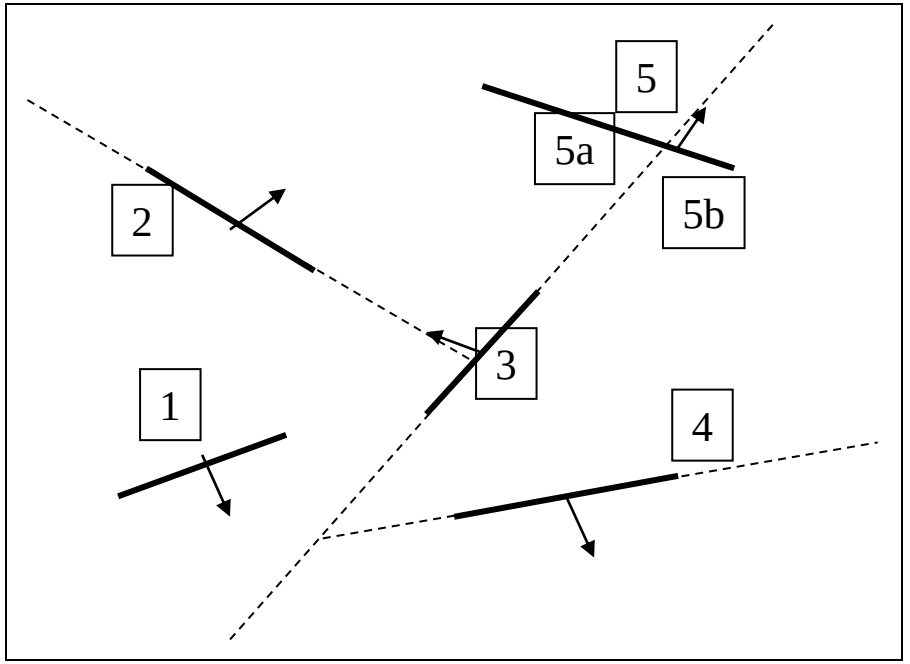
•Choose polygon arbitrarily

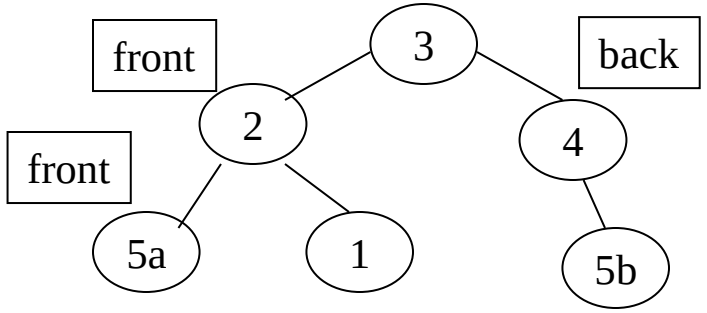
•Divide scene into front (relative to normal) and back half-spaces.

•Split any polygon lying on both sides.

•Choose a polygon from each side – split scene again.

•**Recursively divide each side until each node contains only 1 polygon.**





BSP Tree

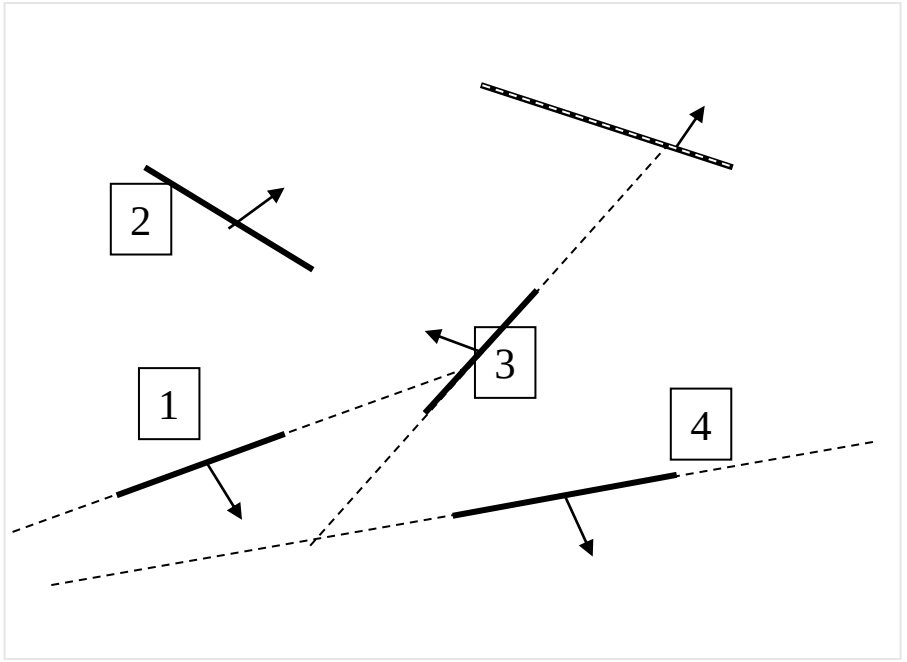
•Choose polygon arbitrarily

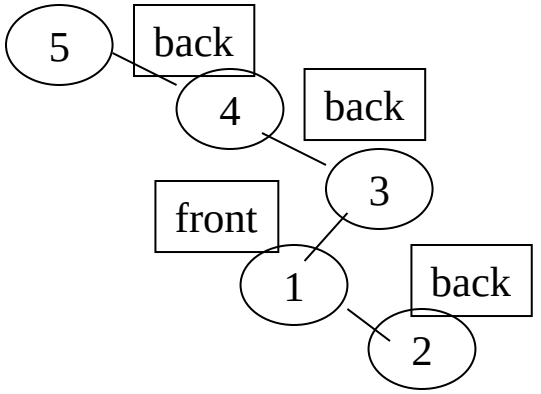
•Divide scene into front (relative to normal) and back half-spaces.

•Split any polygon lying on both sides.

•Choose a polygon from each side – split scene again.

•Recursively divide each side until each node contains only 1 polygon.





Alternate formulation starting at 5

Area-subdivision Method

- This technique for hidden-surface removal is essentially an image-space method, but object-space operations can be used to accomplish depth ordering of surfaces.
- The **area-subdivision method** takes advantage of area coherence in a scene by locating those projection areas that represent part of a single surface.

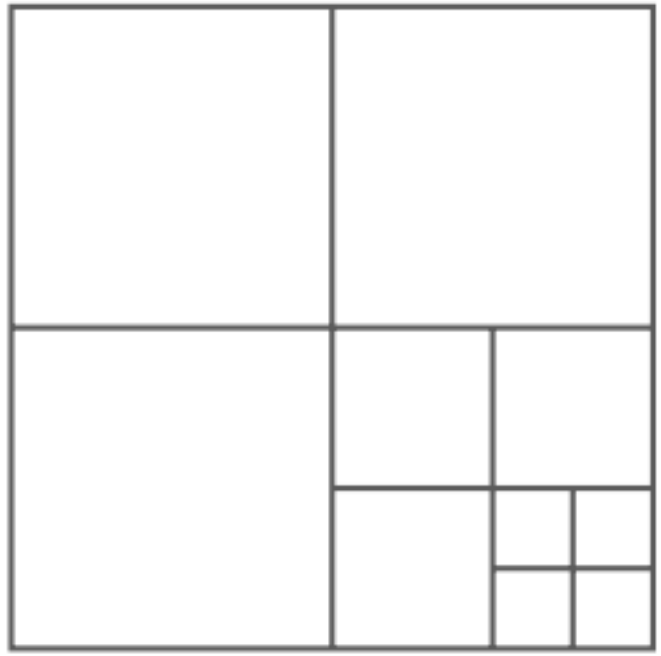


FIGURE 20
Dividing a square area into equal-sized quadrants at each step.

Surrounding Surface: A surface that completely encloses the area.

Overlapping Surface: A surface that is partly inside and partly outside the area.

Inside Surface: A surface that is completely inside the area.

Outside Surface: A surface that is completely outside the area.

Condition 1: An area has no inside, overlapping, or surrounding surfaces (all surfaces are outside the area).

Condition 2: An area has only one inside, overlapping, or surrounding surface.

Condition 3: An area has one surrounding surface that obscures all other surfaces within the area boundaries.

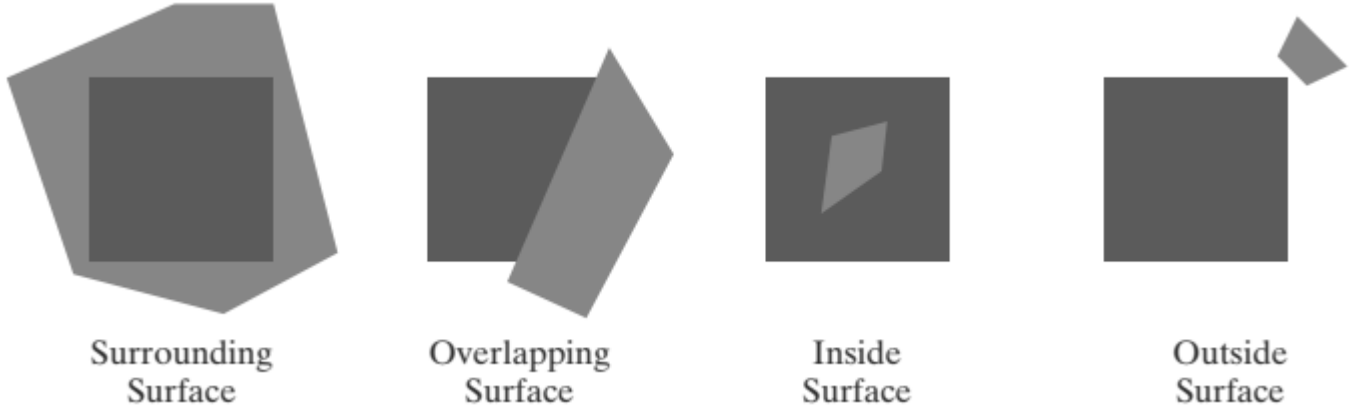
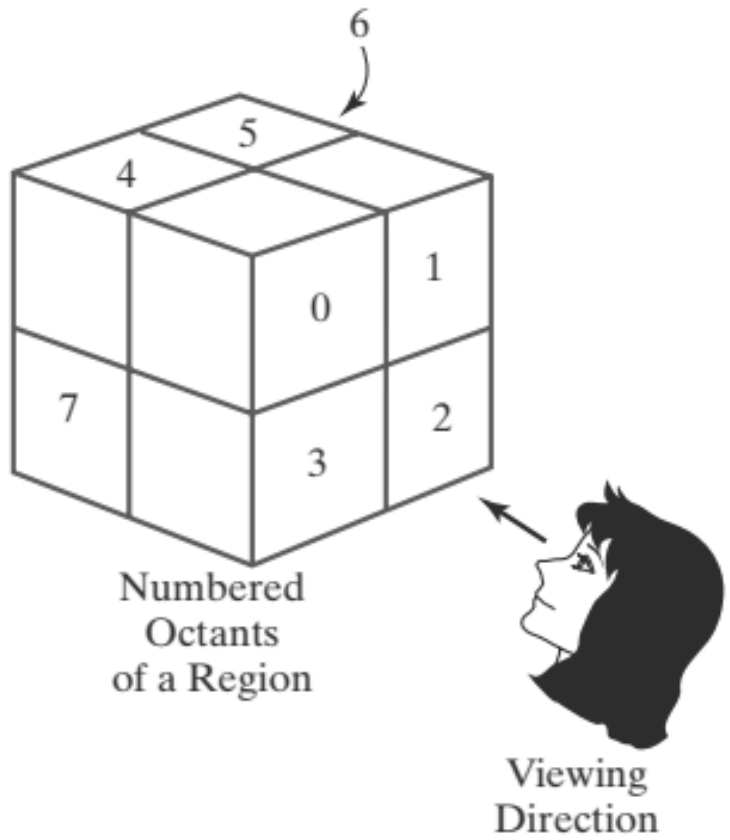


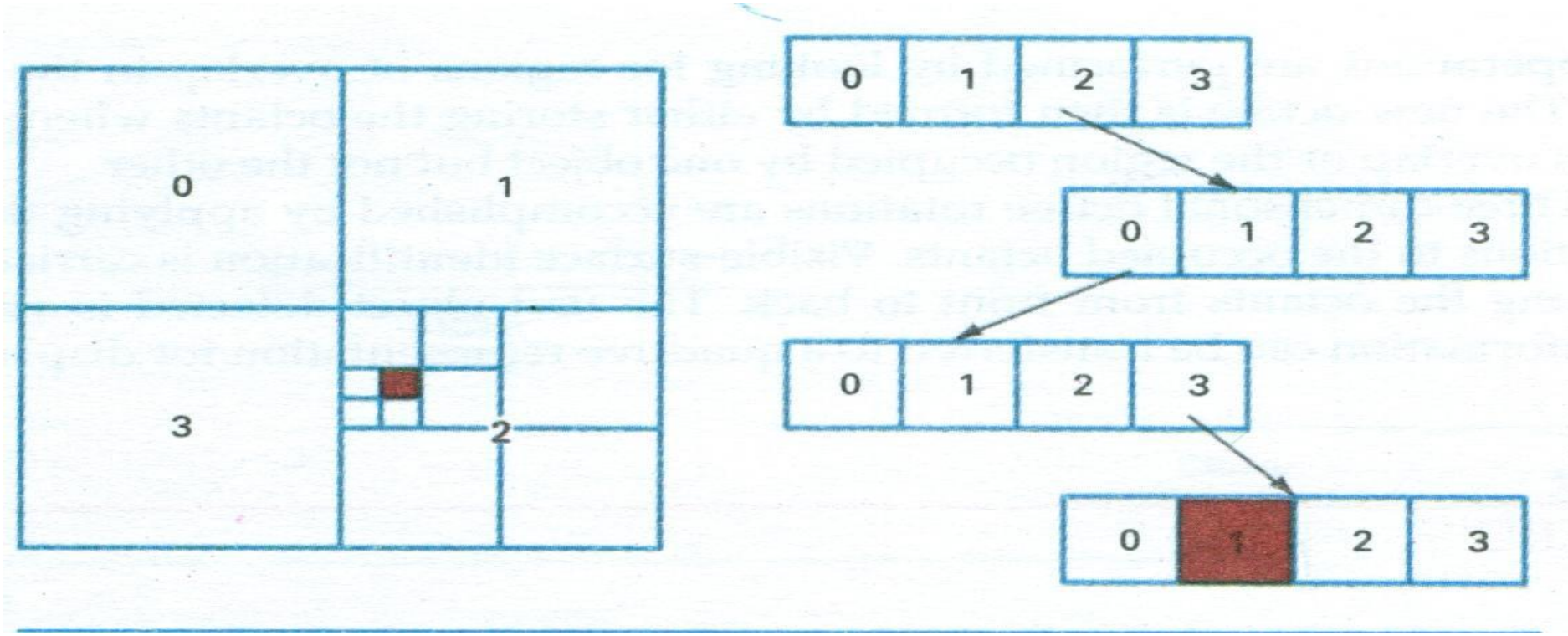
FIGURE 21
Possible relationships between polygon surfaces and a rectangular section of the viewing plane.

Octrees

- Hierarchical tree structure to represent solid objects
- Each node corresponds to a region of three-dimensional space.
- Octree representation is commonly used in Medical Imaging & other applications that require display of object cross-sections.
- Extension of Quadtree encoding in 2D space.

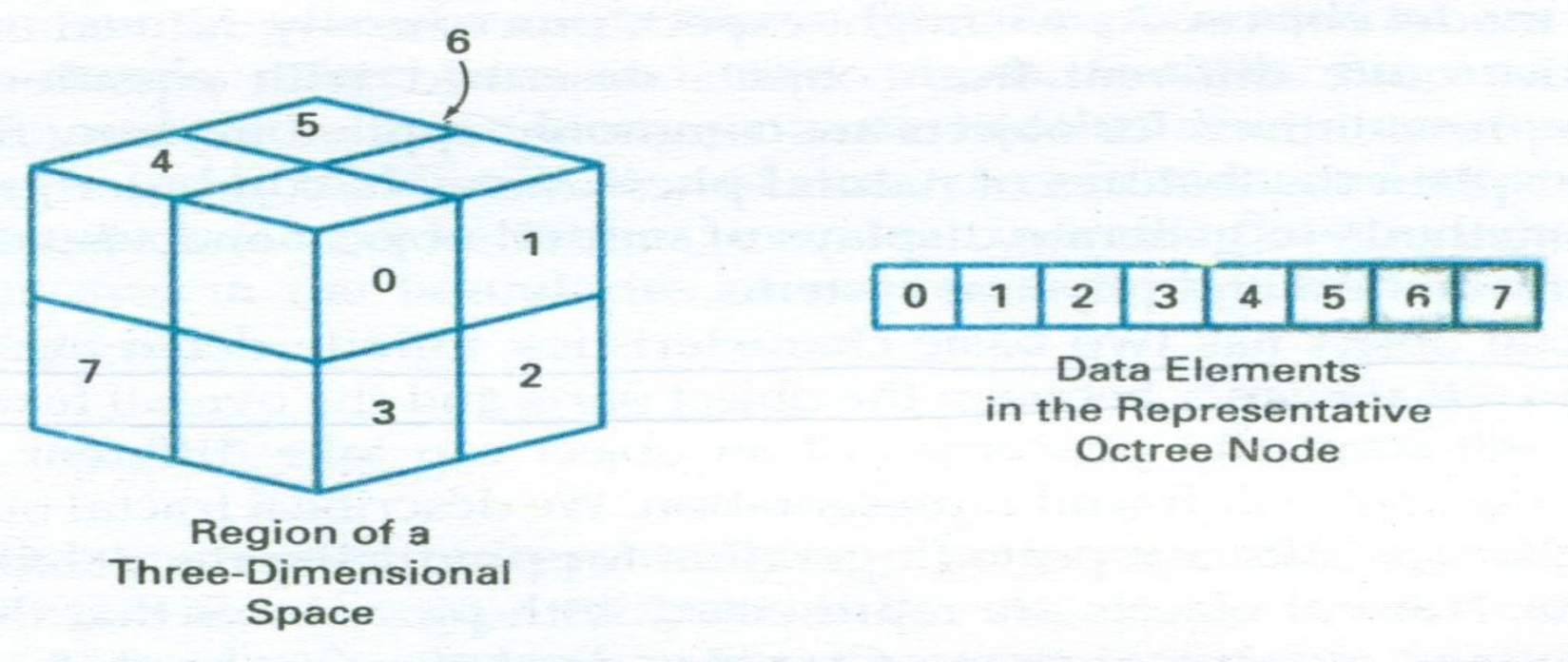


Quadtree representation



Octree representation

- Various manipulation routines can be applied to the solid (e.g. union, intersection or difference)



Shading Methods

- Shading is referred to as the implementation of the illumination model at the pixel points or polygon surfaces of the graphics objects.
- Shading model is used to compute the intensities and colors to display the surface. The shading model has two primary ingredients: properties of the surface and properties of the illumination falling on it.
- The principal surface property is its reflectance, which determines how much of the incident light is reflected. If a surface has different reflectance for the light of different wavelengths, it will appear to be colored.
- An object illumination is also significant in computing intensity. The scene may have to save illumination that is uniform from all direction, called diffuse illumination.

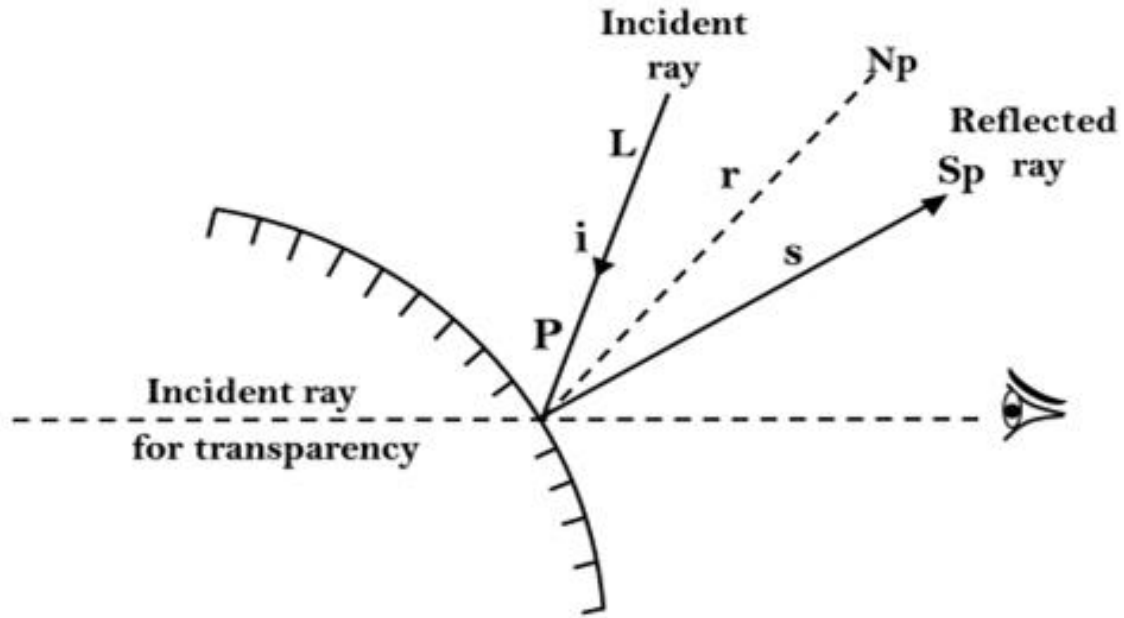
- The simplest form of shading considers only diffuse illumination:

$$E_{pd}=R_p\, I_d$$

- where E_{pd} is the energy coming from point P due to diffuse illumination. I_d is the diffuse illumination falling on the entire scene, and R_p is the reflectance coefficient at P which ranges from shading contribution from specific light sources will cause the shade of a surface to vary as to its orientation concerning the light sources changes and will also include specular reflection effects. In the above figure, a point P on a surface, with light arriving at an angle of incidence i, the angle between the surface normal N_p and a ray to the light source. If the energy I_{ps} arriving from the light source is reflected uniformly in all directions, called diffuse reflection, we have

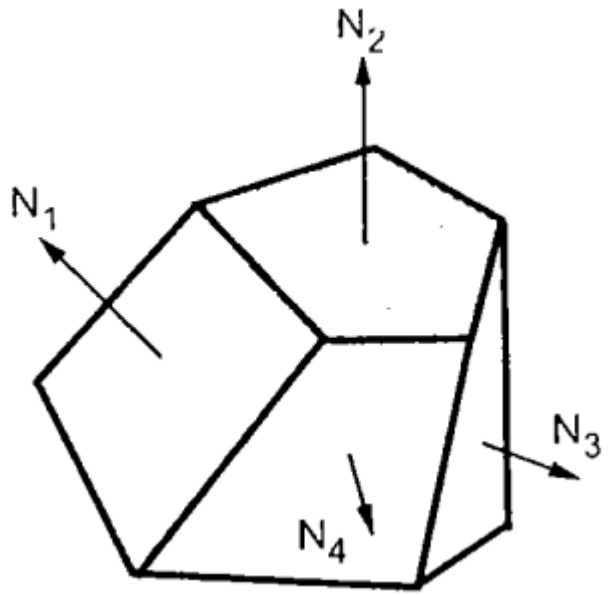
$$E_{ps}=(R_p \cos i)I_{ps}$$

- This equation shows the reduction in the intensity of a surface as it's tipped obliquely to the light source.



Constant Intensity Shading

- A fast and straightforward method for rendering an object with polygon surfaces is constant intensity shading, also called Flat Shading.
- In this method, a single intensity is calculated for each polygon. All points over the surface of the polygon are then displayed with the same intensity value.
- Constant Shading can be useful for quickly displaying the general appearances of the curved surface.



Gouraud Shading

- This Intensity-Interpolation scheme, developed by Gouraud and usually referred to as Gouraud Shading, renders a polygon surface by linear interpolating intensity value across the surface.
- Intensity values for each polygon are coordinate with the value of adjacent polygons along the common edges, thus eliminating the intensity discontinuities that can occur in flat shading.
- Each polygon surface is rendered with Gouraud Shading by performing the following calculations:

1. Determining the average unit normal vector at each polygon vertex.
2. Apply an illumination model to each vertex to determine the vertex intensity.
3. Linear interpolate the vertex intensities over the surface of the polygon.

Phong Shading

- A more accurate method for rendering a polygon surface is to interpolate the normal vector and then apply the illumination model to each surface point.
- This method developed by Phong Bui Tuong is called Phong Shading or normal vector Interpolation Shading. It displays more realistic highlights on a surface and greatly reduces the Match-band effect.
- A polygon surface is rendered using Phong shading by carrying out the following steps:

1. Determine the average unit normal vector at each polygon vertex.
2. Linearly & interpolate the vertex normals over the surface of the polygon.
3. Apply an illumination model along each scan line to calculate projected pixel intensities for the surface points.

Global Illumination

- Illumination models in computer graphics are often approximations of the physical laws that describe surface-lighting effects.
- To reduce computations, most packages use empirical models based on simplified photometric calculations.
- Surface rendering is performed through calculating the interaction of an object's surface with the light striking it.
- This type of illumination model is known as *local illumination*, and considers only the properties of that object and the light that directly strikes it.
- To produce more realistic lighting effects, we must also consider the contribution of light that is reflected from other objects onto the surface of the object being shaded.
- This type of illumination, called *global illumination*, can be more accurate, but that accuracy comes at the expense of additional computation.

Ray Tracing Method

- Some global illumination methods, such as ray-tracing, attempt to determine surface shading by following light rays from the eyepoint back into the scene through the pixels of the image plane.

- *Ray casting* is used in constructive solid geometry for locating surface intersections along a ray from a pixel position.

- Ray casting is also a means for identifying visible surfaces in a scene.

- **Ray tracing** is the generalization of the basic ray-casting procedure.

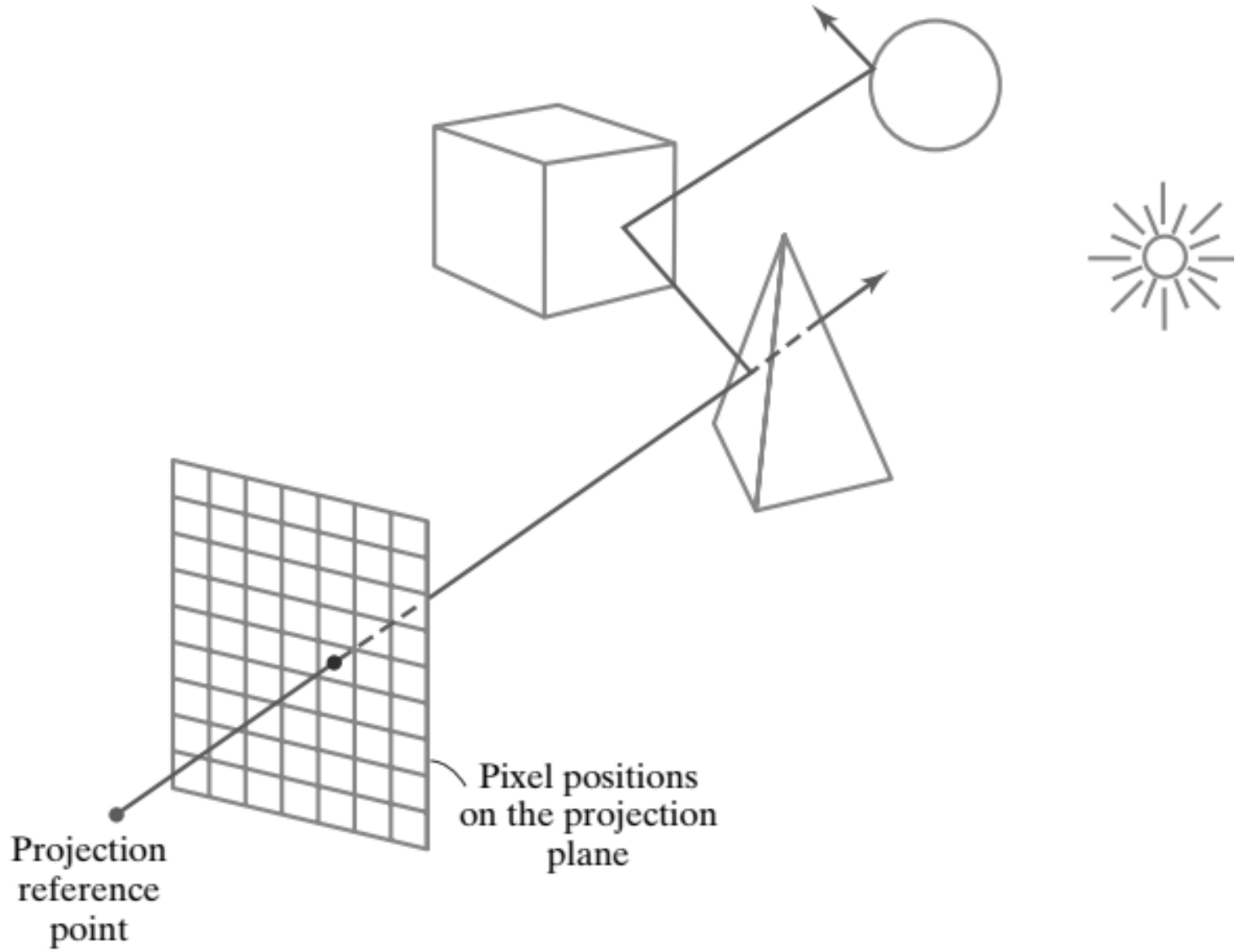


FIGURE 1
Multiple reflection and transmission paths for a ray from the projection reference point through a pixel position and on into a scene containing several objects.

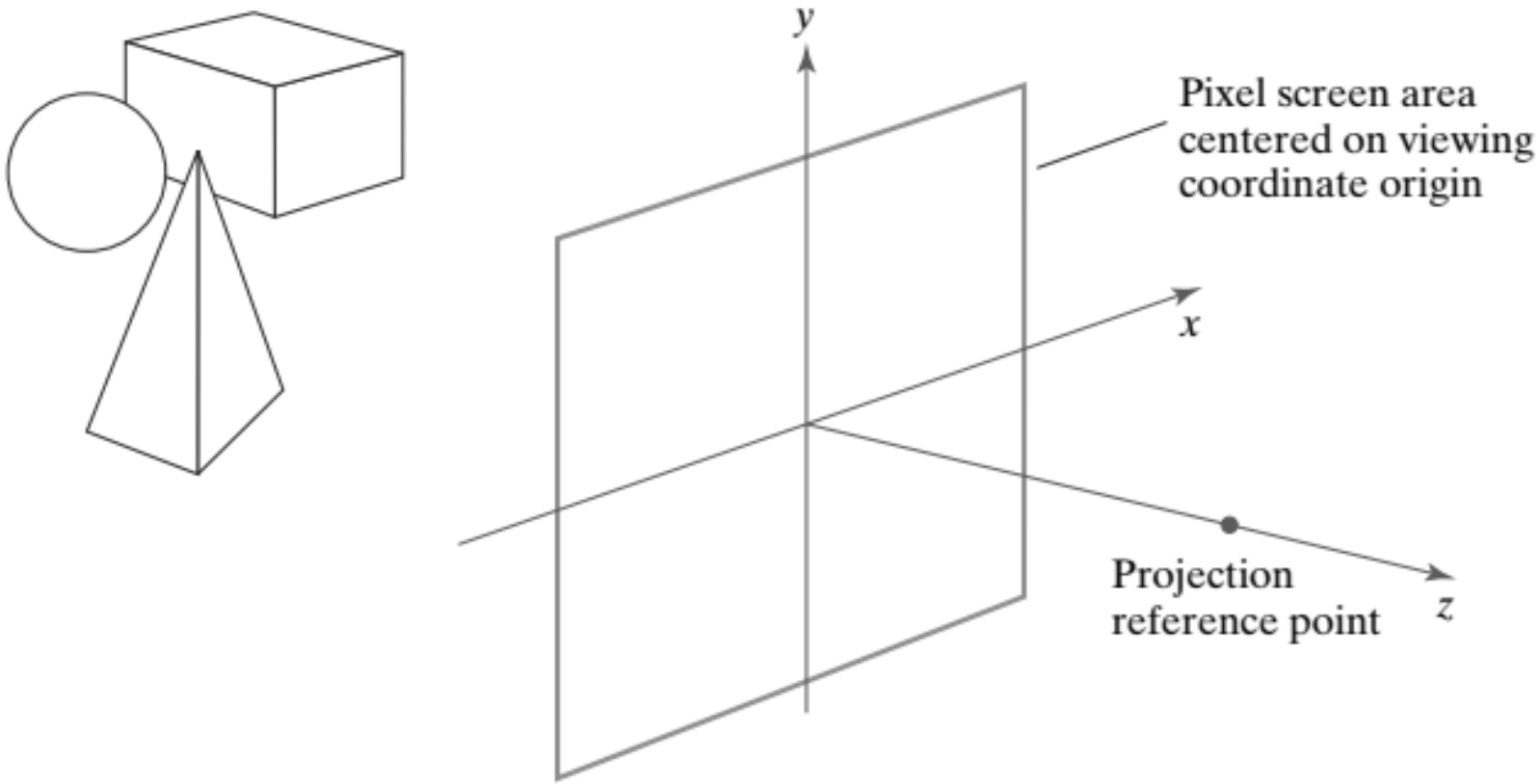


FIGURE 2
Ray-tracing coordinate-reference frame.

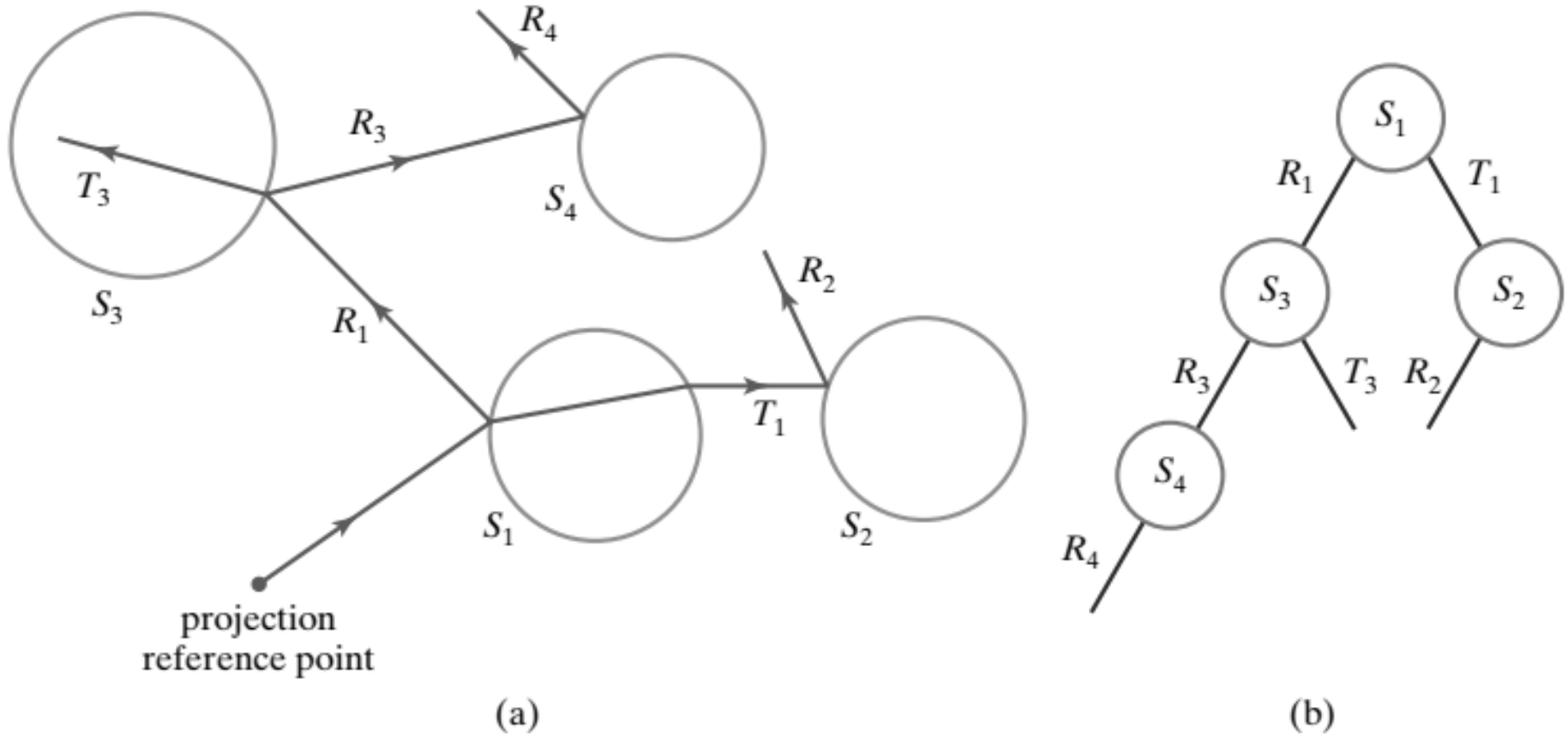


FIGURE 3
The reflection and refraction paths for a pixel ray traveling through a scene are shown in (a), and the corresponding binary ray-tracing tree is given in (b).

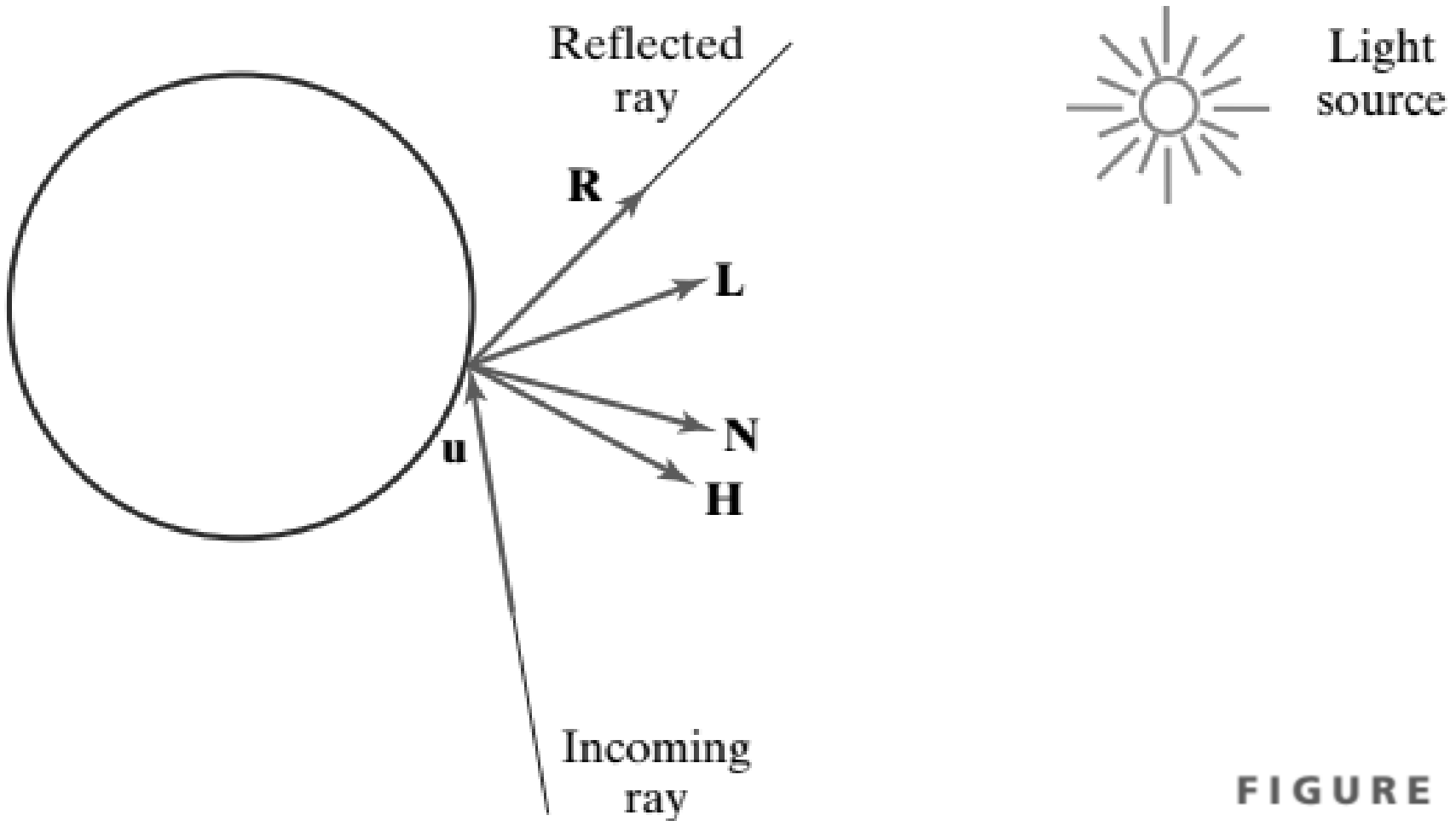


FIGURE 4
Unit vectors at the surface of an object intersected by an incoming ray along direction **u**.

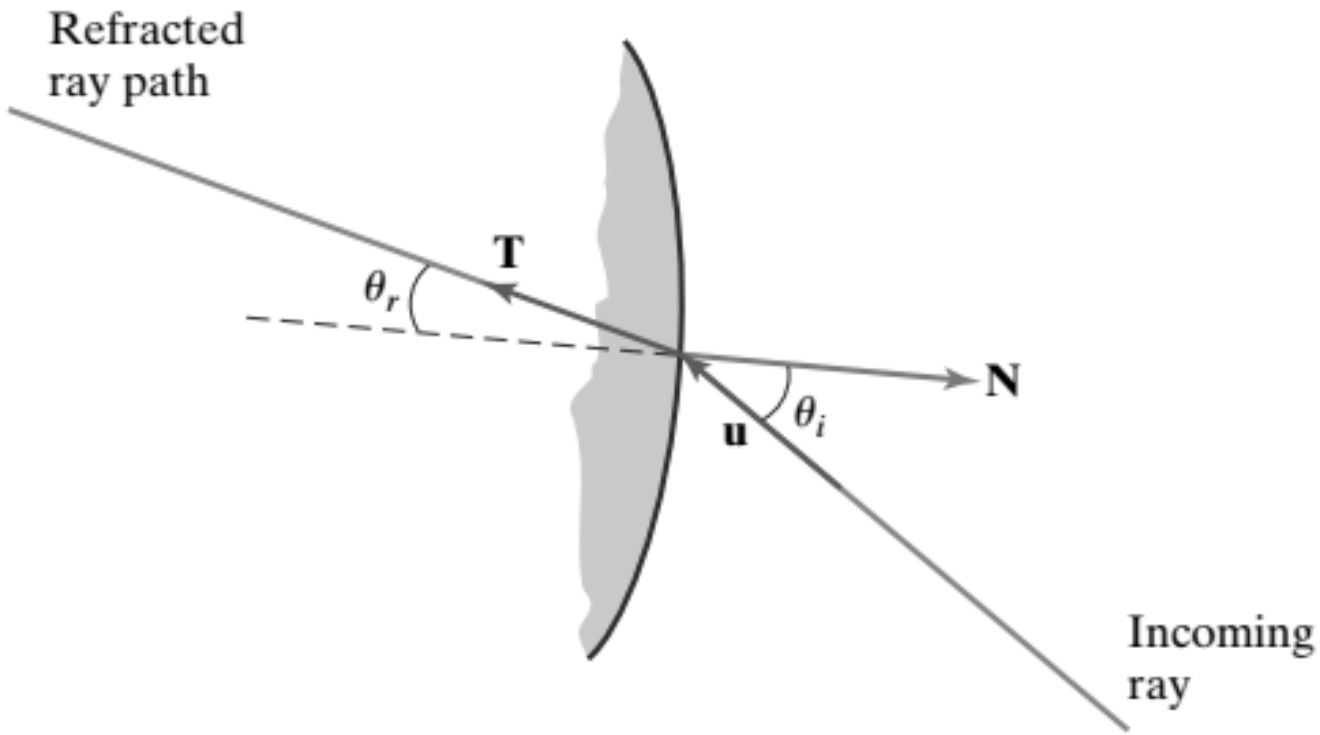


FIGURE 5
Refracted ray-transmission path **T** through a transparent material.

Radiosity Lighting Model

- We can model lighting effects more precisely by considering the physical laws governing the radiant-energy transfers within an illuminated scene. This method for computing pixel color values is generally referred to as the **radiosity model.**

Radiant-Energy Terms

In the quantum model of light, the energy of the radiation is carried by the individual photons. For monochromatic light radiation, the energy of each photon is calculated as

$$E_{\text{photon},f} = hf \tag{24}$$

where the frequency f, measured in hertz (cycles per second), characterizes the color of the light. A blue light has a high frequency within the visible band of the electromagnetic spectrum, and a red light has a low frequency. The frequency also gives the oscillation rate for the amplitude of the electric and magnetic components of the radiation. Parameter h is *Planck's constant*, which has the value 6.6262×10^{-34} joules • sec, independent of the light frequency.

The total energy for monochromatic light radiation is

$$E_f = \sum_{\text{all photons}} hf \tag{25}$$

The radiant energy at a particular light frequency is also referred to as a **spectral radiance.** However, any actual light radiations, even those from a "monochromatic" source, contain a range of frequencies. Therefore, the total radiant energy is the sum over all photons of all frequencies:

$$E = \sum_{f} \sum_{\text{all photons}} hf \tag{26}$$

The amount of radiant energy transmitted per unit of time is called the **radiant flux** Φ:

$$\Phi = \frac{dE}{dt} \tag{27}$$

Radiant flux is also referred to as **radiant power,** and it is measured in watts (joules per second).

To obtain the lighting effects for surfaces in a scene, we calculate the radiant flux per unit area that is leaving a surface. This quantity is called the **radiosity** B, or **radiant exitance**:

$$B = \frac{d\Phi}{dA} \tag{28}$$

The intensity I for the diffuse radiation in direction (θ, ϕ) can be described as the radiant energy per unit time per unit projected area per unit solid angle, or

$$I = \frac{dB}{d\omega \cos\phi} \tag{29}$$

Assuming the surface is an ideal diffuse reflector, we can set the intensity I to a constant for all viewing directions. Thus, $dB/d\omega$ is proportional to the to the projected surface area (Figure 27). To obtain the total rate of energy radiation from the surface point, we need to sum the radiation for all directions.

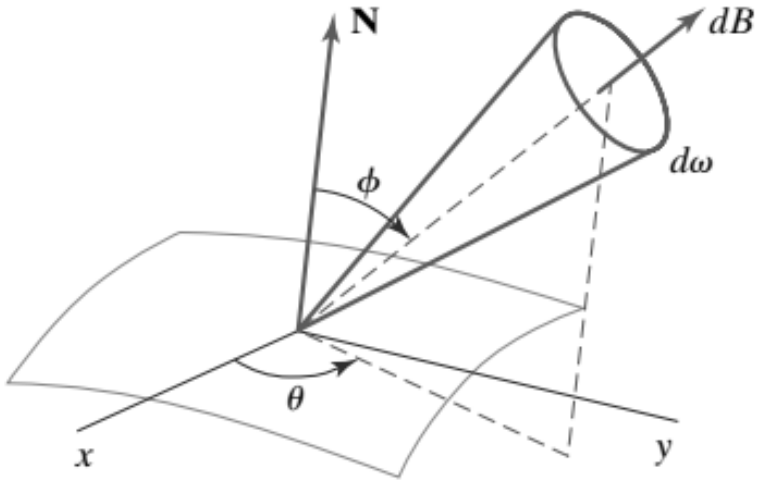


FIGURE 26
Visible radiant energy emitted from a surface point in direction (θ, ϕ) within solid angle $d\omega$.

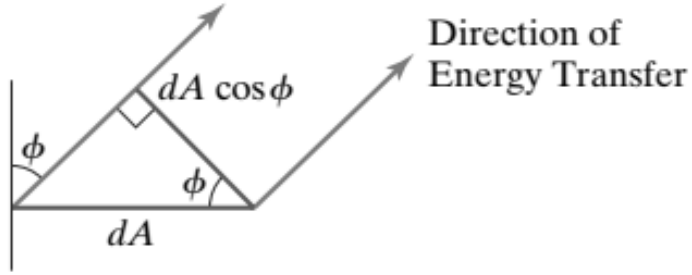


FIGURE 27
For a unit surface element, the projected area perpendicular to the direction of energy transfer is equal to $\cos\phi$.

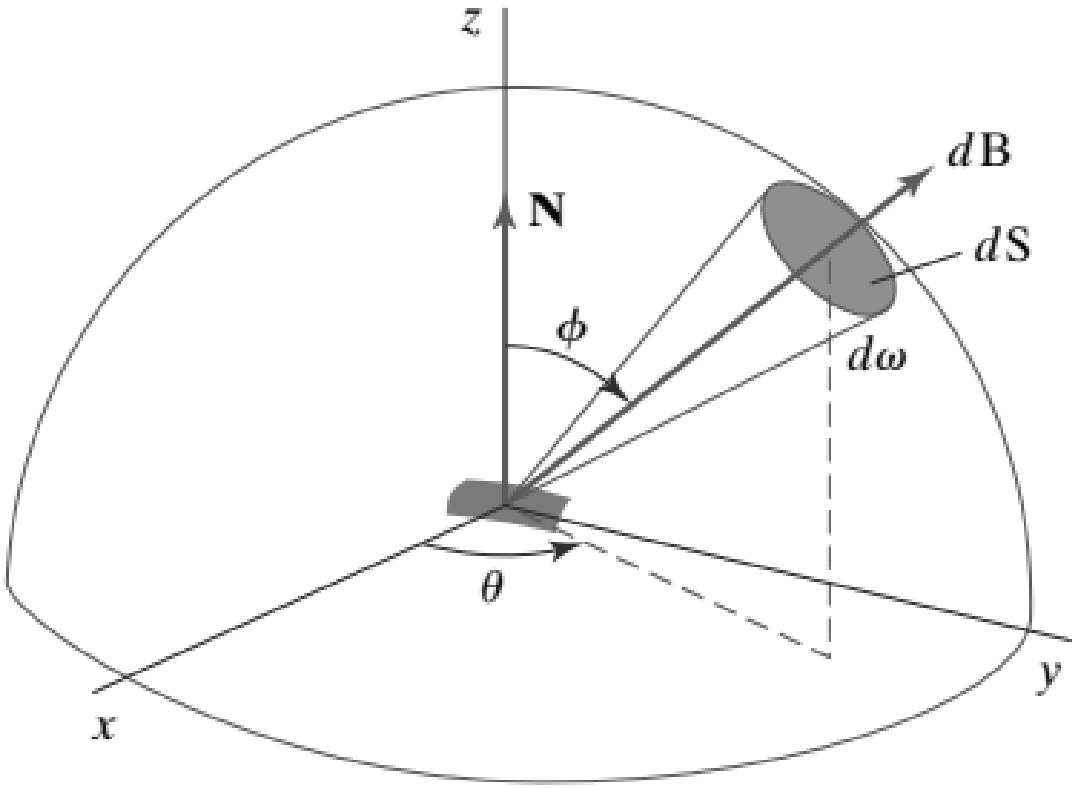


FIGURE 28
Total radiant energy from a surface point is the sum of the contributions in all directions over a hemisphere centered on that surface point.

For a perfect diffuse reflector, I is a constant, so we can express radiant flux B as

$$B = I \int_{\text{hemi}} \cos\phi \, d\omega \tag{31}$$

Also, the differential element of solid angle $d\omega$ can be expressed as

$$d\omega = \frac{dS}{r^2} = \sin\phi \, d\phi \, d\theta$$

so that

$$\begin{aligned} B &= I \int_0^{2\pi} \int_0^{\pi/2} \cos\phi \sin\phi \, d\phi \, d\theta \\ &= I\pi \end{aligned} \tag{32}$$

For a scene with n surfaces in the enclosure, the radiant energy from surface k is described with the following **radiosity equation**:

$$\begin{aligned} B_k &= E_k + \rho_k H_k \\ &= E_k + \rho_k \sum_{j=1}^{n} B_j \, F_{jk} \end{aligned} \tag{34}$$

GPU Architecture

Contents:

- Introduction of CUDA.
- Programmable Model / Parallel Computing with CUDA.
- CUDA Application Programmable Interface.
- Hardware Implementation.
- Performance Guide / Memory Optimizations.
- Introduction of Multi-GPU Programming.

Introduction to CUDA:

- **The Graphics Processor Unit (GPU) as a Data-Parallel Computing Device:**

  GPU: It is a specialized circuit designed to rapidly manipulate and alter memory in such a way, to accelerate the building of images in a frame buffer intended for output to a display.

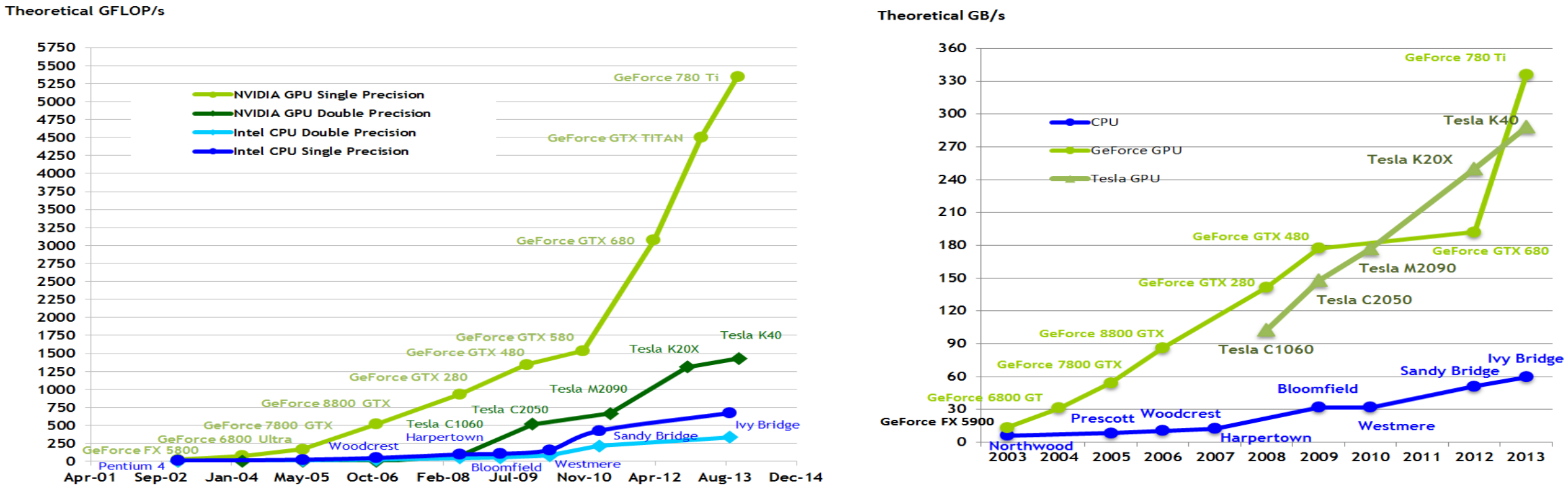


Fig.1. Floating-point operations per second

i. GPU is specialized for compute-intensive, highly parallel computation.

ii. It is designed more transistors are devote to data processing.

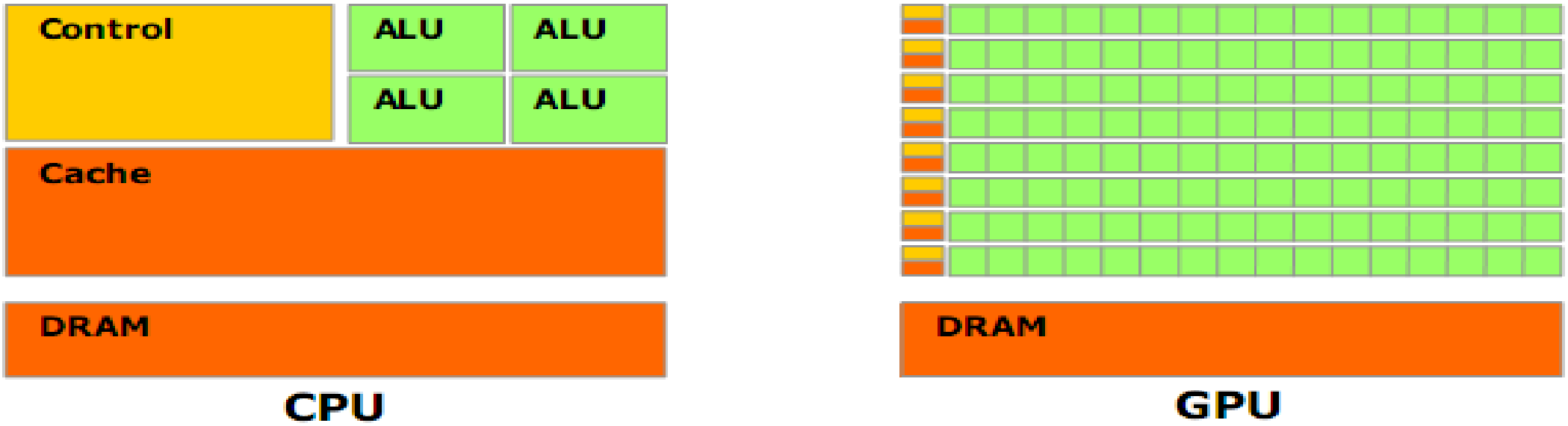


Fig.2. The GPU Devotes More Transistors to Data Processing

GPU Computing Applications

Libraries and Middleware

CUFFT CUBLAS CURAND CUSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX	iray	MATLAB Mathematica

Programming Languages

C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)

CUDA-Enabled NVIDIA GPUs

Kepler Architecture (compute capabilities 3.x)	GeForce 600 Series	Quadro Kepler Series	Tesla K20 Tesla K10
Fermi Architecture (compute capabilities 2.x)	GeForce 500 Series GeForce 400 Series	Quadro Fermi Series	Tesla 20 Series
Tesla Architecture (compute capabilities 1.x)	GeForce 200 Series GeForce 9 Series GeForce 8 Series	Quadro FX Series Quadro Plex Series Quadro NVS Series	Tesla 10 Series

Many Core GPU-Block Diagram

- G80 (launched Nov 2006 – GeForce 8800 GTX)
- 128 Thread Processors execute kernel threads.
- Up to 12,288 parallel threads active.
- Per-block shared memory (PBSM) accelerate processing.

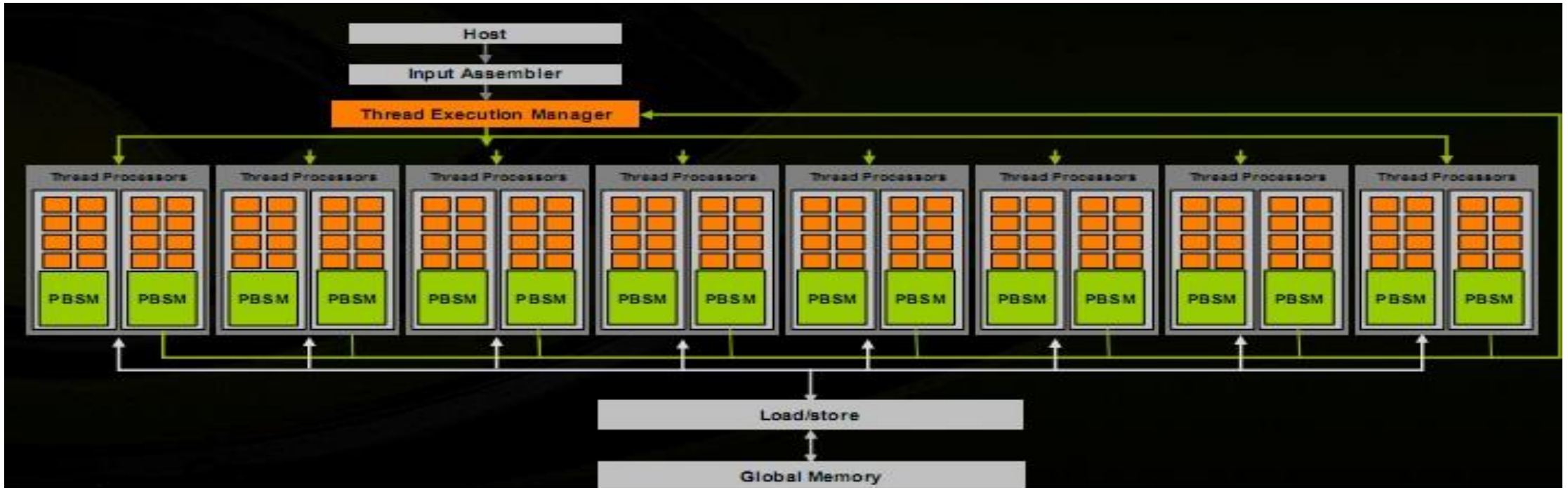


Fig.3. Block Diagram of many core GPU

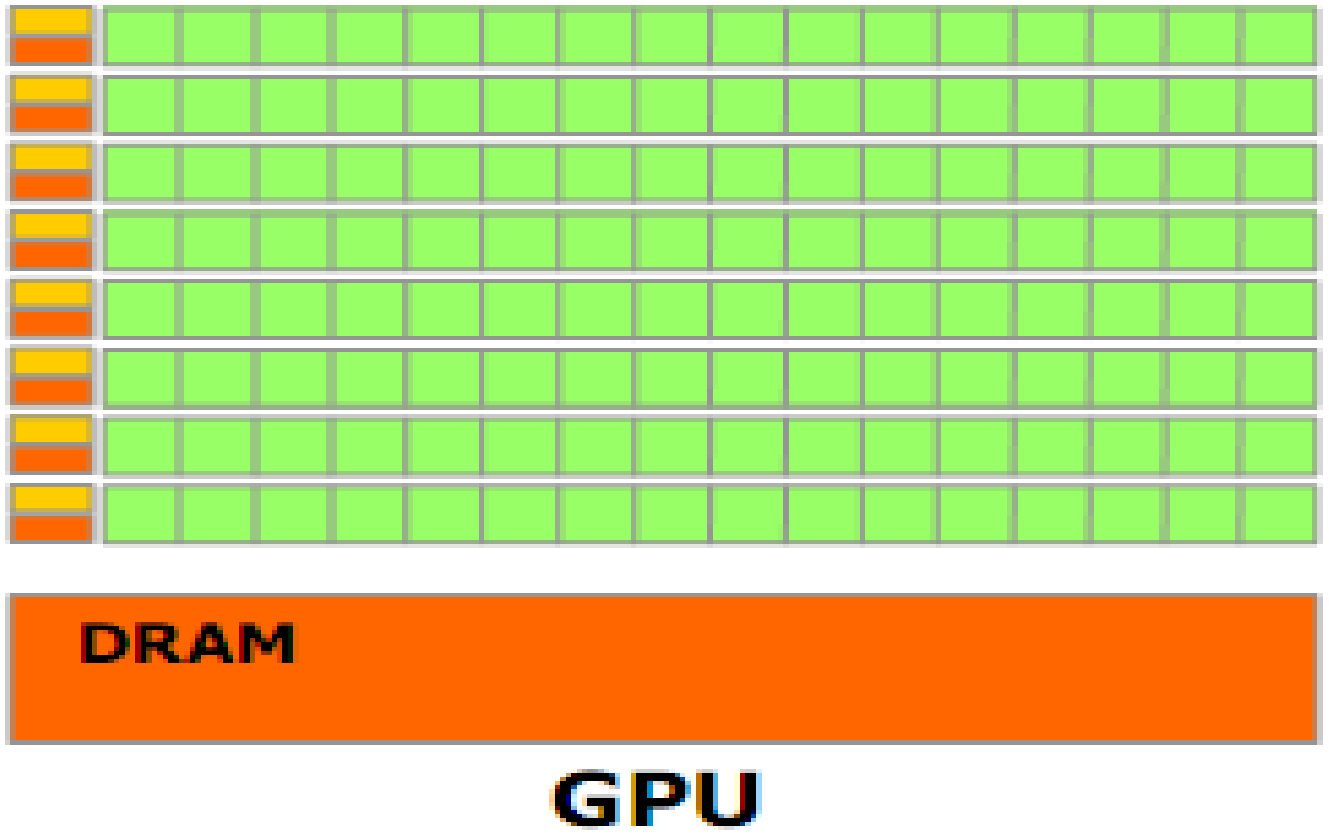


Fig.4. GPU architecture

CUDA GPU Roadmap

Heterogeneous Computing with CUDA:

- CUDA C programming involves running code on two different platforms concurrently: a host system with one or more CPUs and one or more devices with CUDA enabled NVIDIA GPUs.

- NVIDIA devices are associated with rendering graphics.

- Powerful arithmetic engines capable of running thousands of lightweight threads in parallel.

- Difference between Host and Device:

i. Threading resources

ii. Threads

iii. RAM

Difference between Host and Device:

- **Threading resources:** Execution pipelines on host systems can support a limited number of concurrent threads.
  - Four quad-core processors can run only 16 threads concurrently.
  - All NVIDIA GPUs can support at least 768 concurrently active threads per multiprocessor.
  - NVIDIA GeForce GTX 280 can support more than 30,000 active threads.

- **Threads:** Threads on a CPU are generally heavyweight entities. The operating system must swap threads on and off of CPU execution channels to provide multithreading capability. Threads on GPU are extremely lightweight.

- **RAM:** Both the host system and the device have RAM.
  - On the host system, RAM is generally equally accessible to all code.
  - On the device, RAM is divided virtually and physically into different types.

Maximum Performance Benefit:

- **High Priority:** To get the maximum benefit from CUDA, focus first on finding ways to parallelize sequential code.

- Amdahl's law specifies the maximum speed-up.

- **Amdahl's Law:** also known as **Amdahl's argument**.

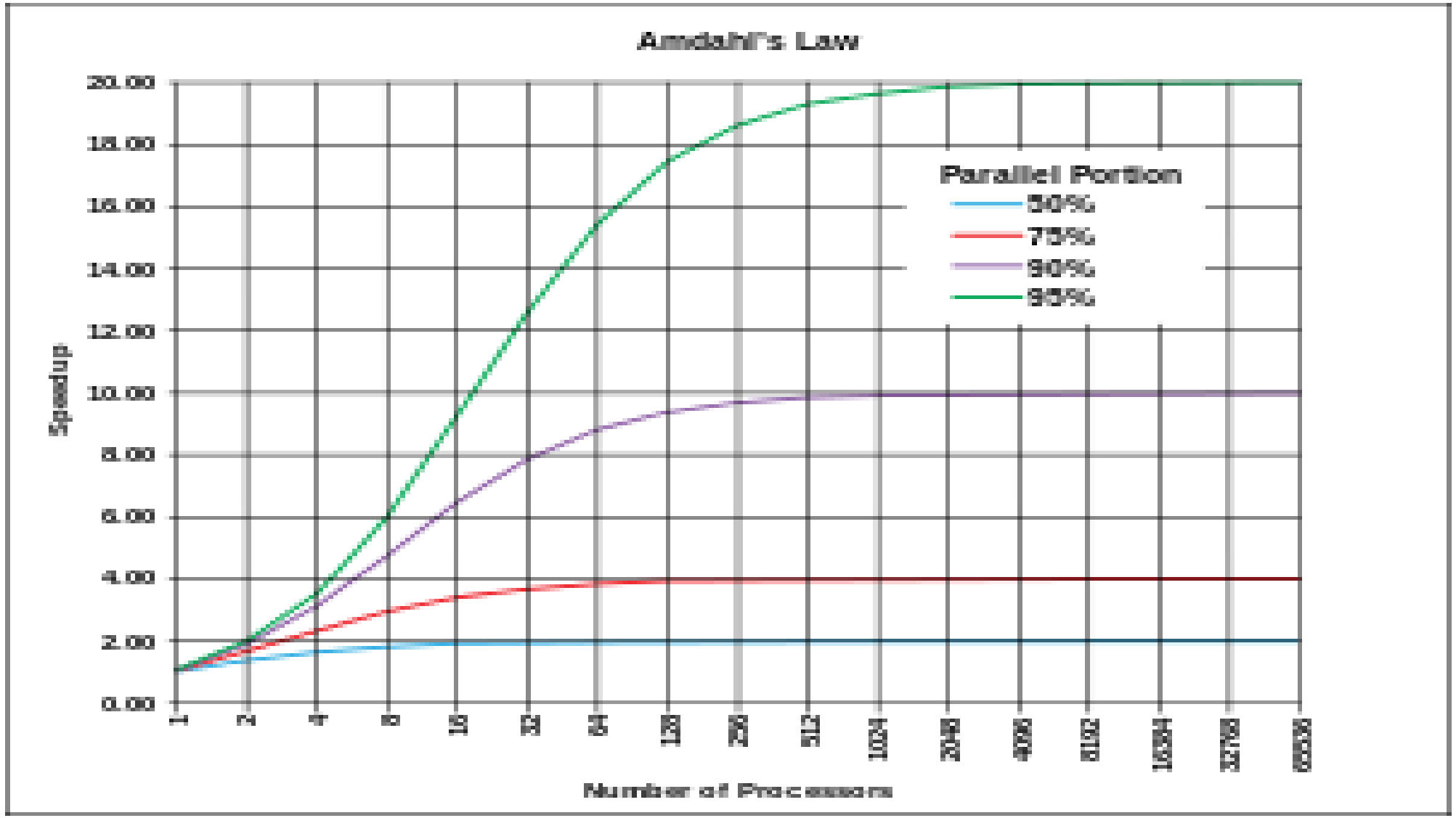
➢ This law is used to find the maximum expected improvement to an overall system when only part of the system is improved.

➢ It is often used in parallel computing to predict the theoretical maximum speedup using multiple processors.

➢ The maximum speed-up (S) of a program is

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

where, P is the fraction of the total serial execution time taken by the portion of code. N is the number of processors.


Amdahl's Law
Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
Parallel Portion
50%
75%
90%
95%
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors

Understanding the Programming Environment:

- **CUDA Compute Capability:** It describes:

1. Features of hardware.
2. Set of instructions supported by the device.
3. Maximum number of threads per block.
4. Number of register per multiprocessor.

The compute capability of the GPU can be queried programmatically in the CUDA SDK in the *deviceQuery* sample.

The information is obtained by calling *cudaGetDeviceProperties()*.

Fig.7. Sample CUDA configuration data reported by deviceQuery

- **C Runtime for CUDA and Driver API Version:**

  The CUDA driver API and the C runtime for CUDA are two of the programming interface to CUDA.

  Version number is important because the CUDA driver API is backward compatible but not forward compatible.

- Applications, plug-ins, and libraries (including the C runtime for CUDA) compiled against a particular version of the driver API.

- It will continue to work on subsequent driver releases.

- Applications, plug-ins, and libraries compiled against a particular version of the driver API may not work on earlier versions of the driver.

CUDA: A new architecture for computing on the GPU

- CUDA stands for Compute Unified Device Architecture .
- It is a new hardware and software architecture for issuing and managing computations on the GPU as a data-parallel computing device.
- There is no need of mapping them to a graphics API.
- It is available for the GeForce 9 Series, Quadro FX 5600/4600, and Tesla solutions.
- A hardware driver, an application programming interface (API) and its runtime and two higher-level mathematical libraries of common usage, CUFFT and CUBLAS.
- The hardware has been designed to support lightweight driver and runtime layers, resulting in high performance.

- Developed by NVIDIA in late 2006.
- CUDA is a compiler and toolkit for programming NVIDIA GPUs.
- CUDA API extends the C programming language.
- Runs on thousands of threads.
- It is an scalable model.
- Objectives:
    - Express Parallelism
    - Give high level abstraction from hardware

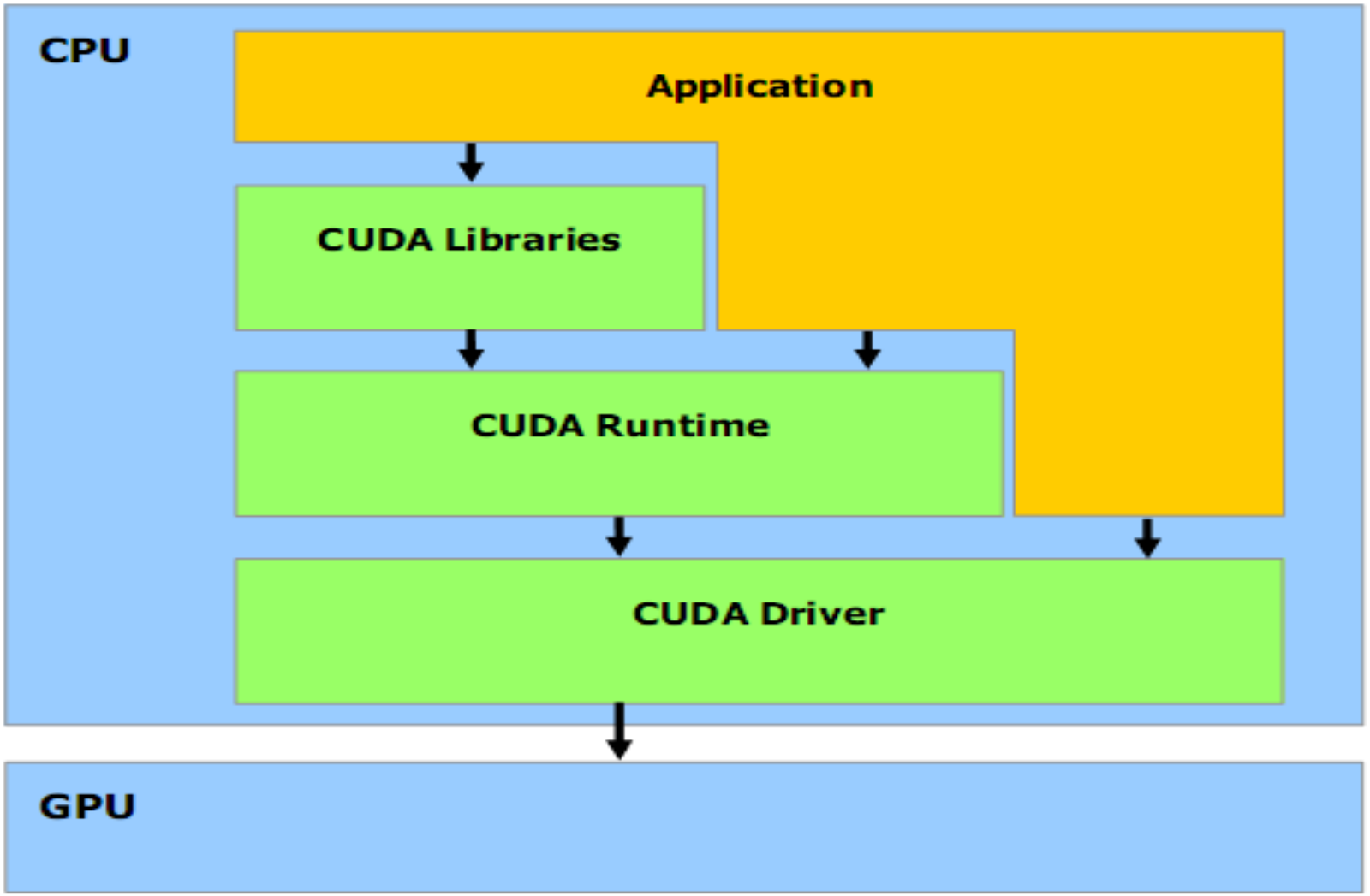


Fig.9. CUDA Software Stack

Systematically Processing Flow on CUDA

- Copy data from main memory to GPU memory.
- CPU instructs the process to GPU.
- GPU execute parallel in each core.
- Copy the result from GPU memory to main memory.

Processing Flow on CUDA

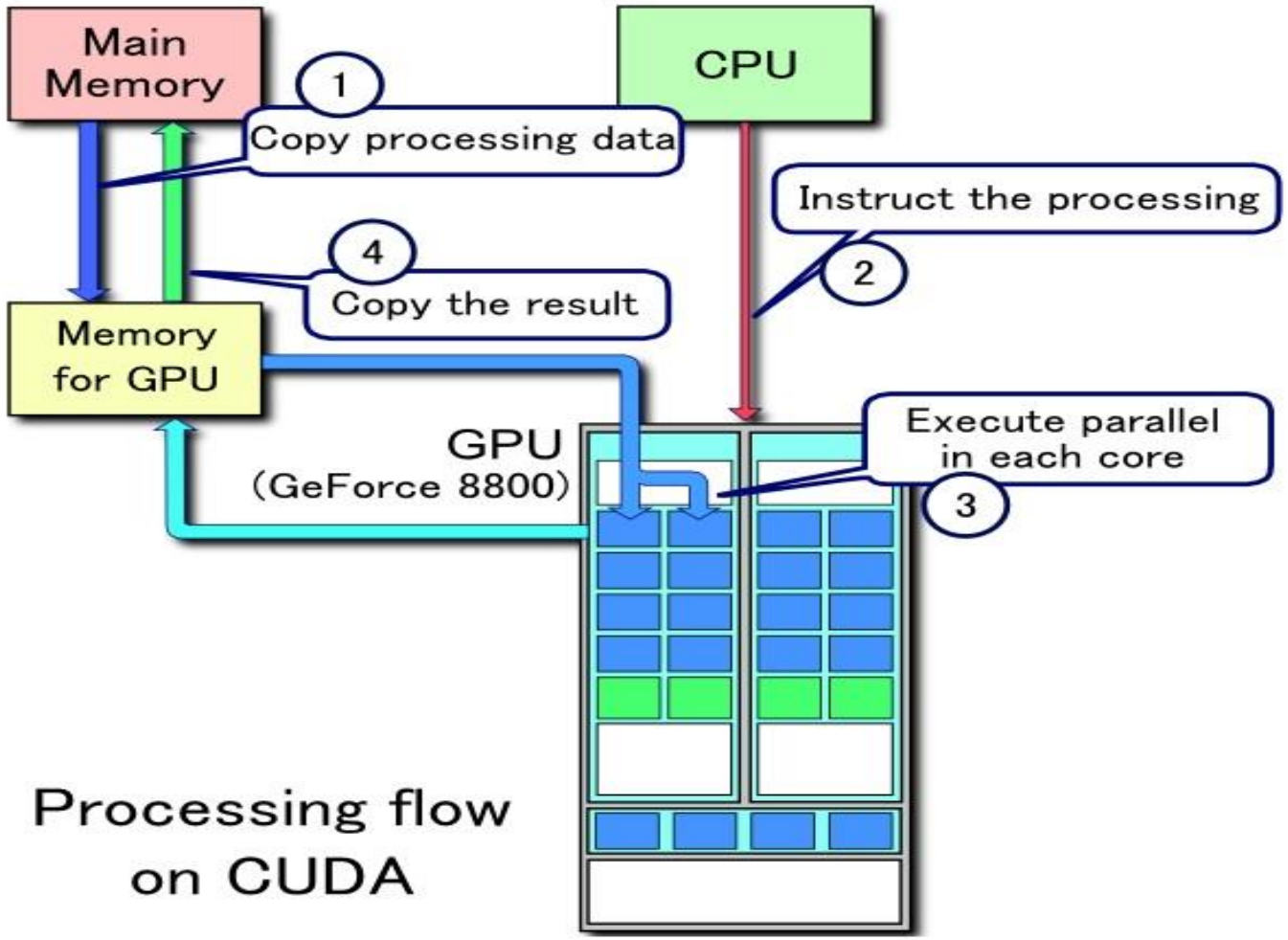


Fig.10. Processing Flow on CUDA

Programming Model: A Highly Multithreaded Coprocessor

GPU is viewed as a compute device operating as a coprocessor to the main CPU (host).

- Data-parallel, compute intensive functions should be off-loaded to the device.
- Functions that are executed many times, but independently on different data, are prime candidates.
  - i.e. body of for-loops
- A function compiled for the device is called a Kernel.
- The kernel is executed on the device as many different threads
- Both host (CPU) and device (GPU) manage their own memory, host memory and device memory.
  - Data can be copied between them

Thread Batching: Grid of thread blocks

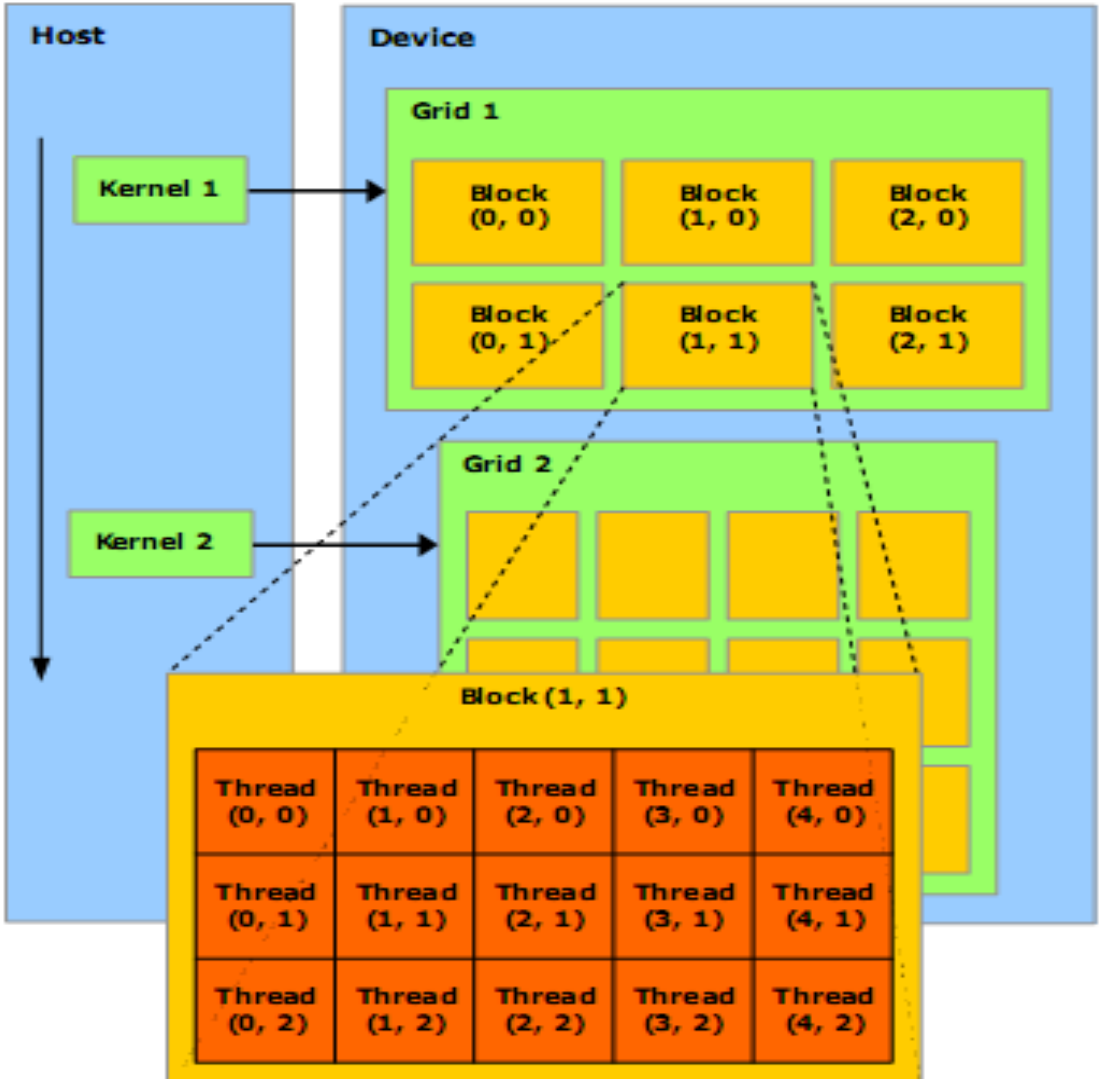


Fig.3. Grid of Thread Blocks

- The computational grid consist of a grid of thread blocks
- Each thread executes the kernel
- The application species the grid and block dimensions
- The grid layouts can be 1, 2, or 3-dimensional
- The maximal sizes are determined by GPU memory and kernel complexity
- Each block has an unique block ID
- Each thread has an unique thread ID (within the block)

Grid of Thread Blocks

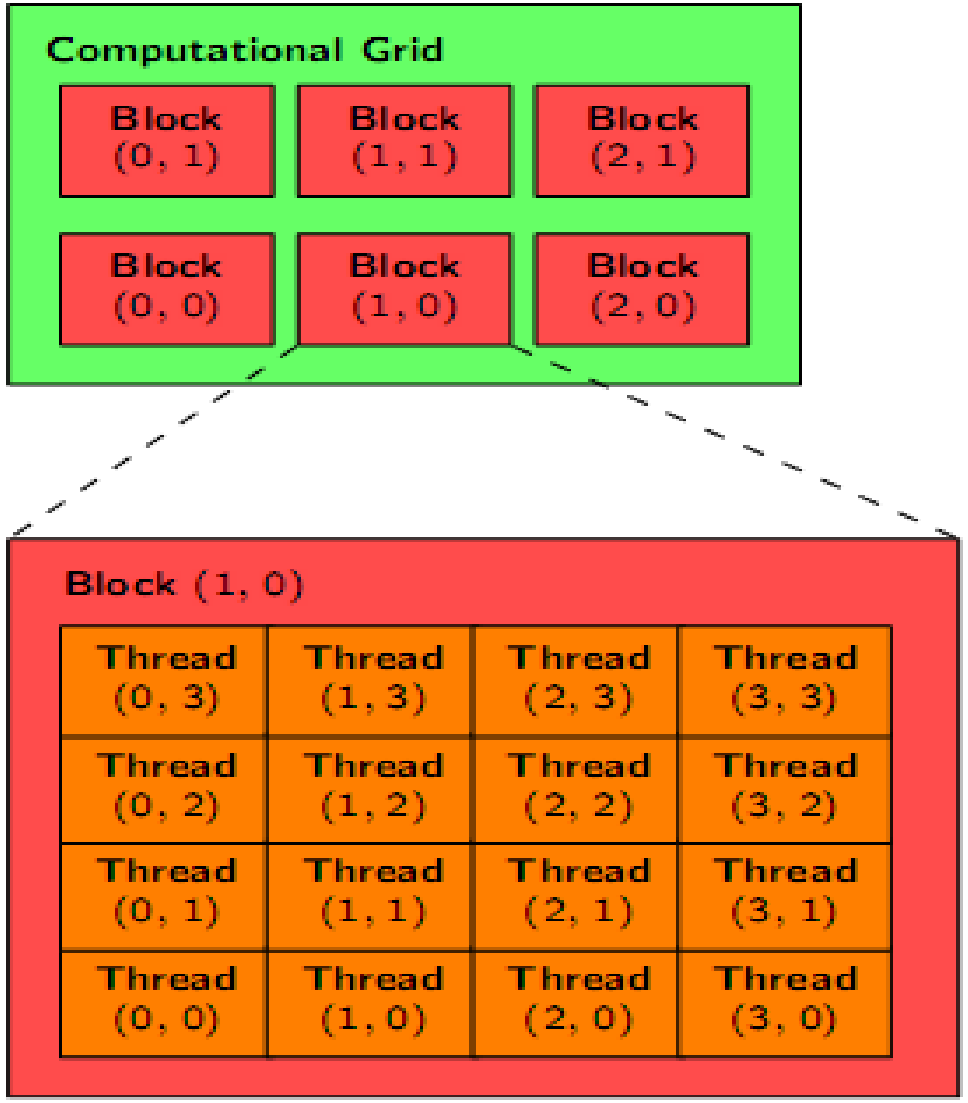


Fig.4. Computational Grid

Some Design Goals

- Enable heterogeneous systems (i.e., CPU+GPU)
  - CPU & GPU are separate devices with separate DRAMs
- Scale to 100’s of cores, 1000’s of parallel threads.
- Programmers focus on parallel algorithms.

Heterogeneous Programming

- CUDA = serial program with parallel kernels, all in C
  - Serial C code executes in a **host** thread (i.e. CPU thread)
  - Parallel kernel C code executes in many **device** threads across multiple processing elements (i.e. GPU threads).

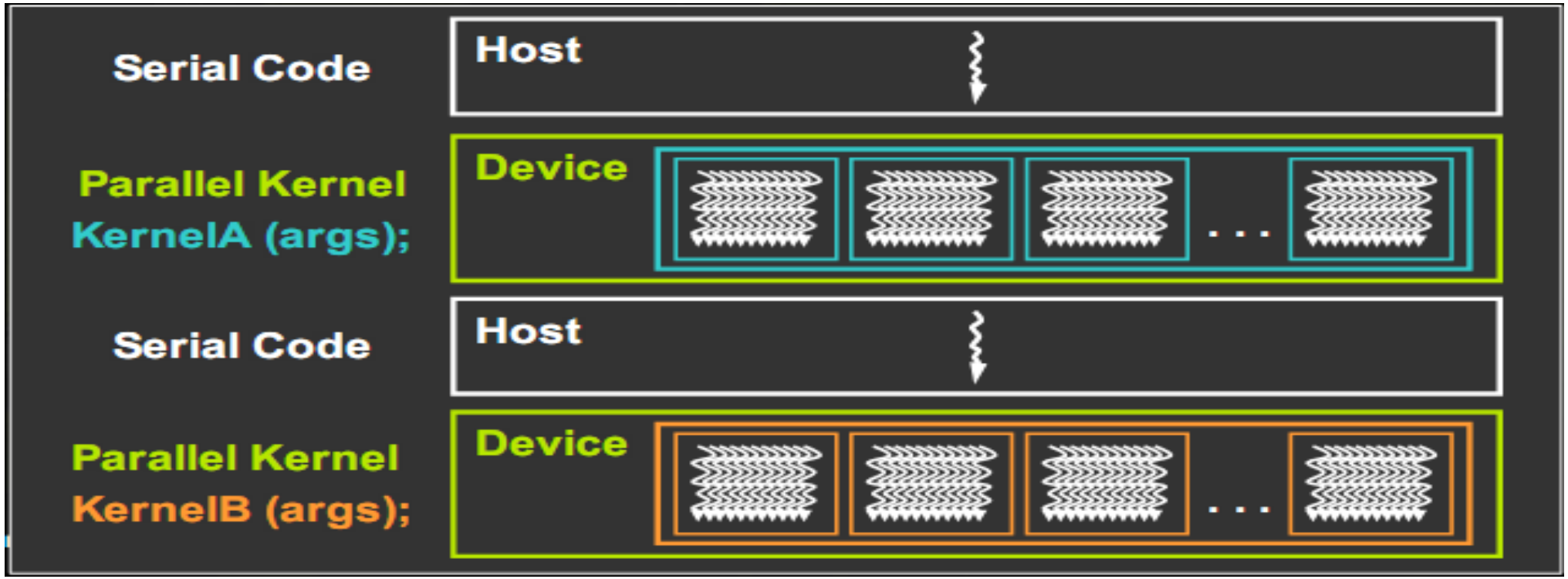


Fig.5. Programming Between Host and Device

Kernel = Many Concurrent Threads

- One kernel is executed at a time on the device
- Many threads execute each kernel
  - Each thread executes the same code…
  - … on different data based on its thread ID

- CUDA threads might be
  - Physical threads
    - As on NVIDIA GPUs
    - GPU thread creation and context switching are essentially free
  - Or Virtual threads
    - E.g. 1 CPU core might execute Multiple CUDA threads

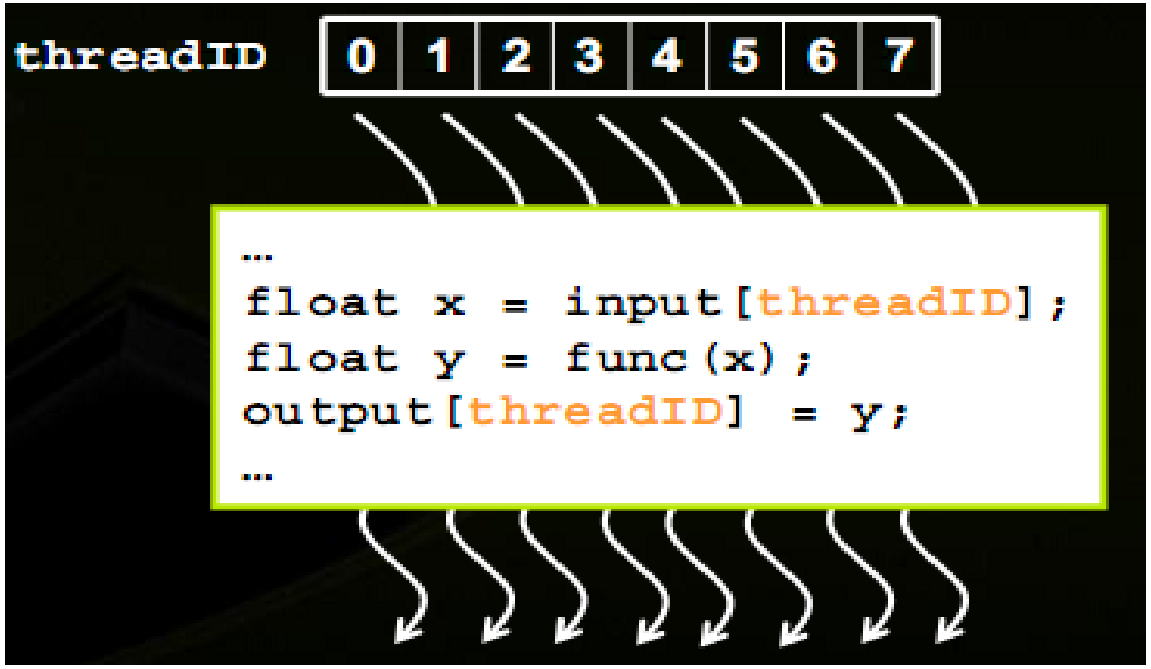


Fig.6. Thread Execution

Hierarchy of Concurrent Threads

- Threads are grouped into **thread blocks**
  - Kernel = **grid** of thread blocks

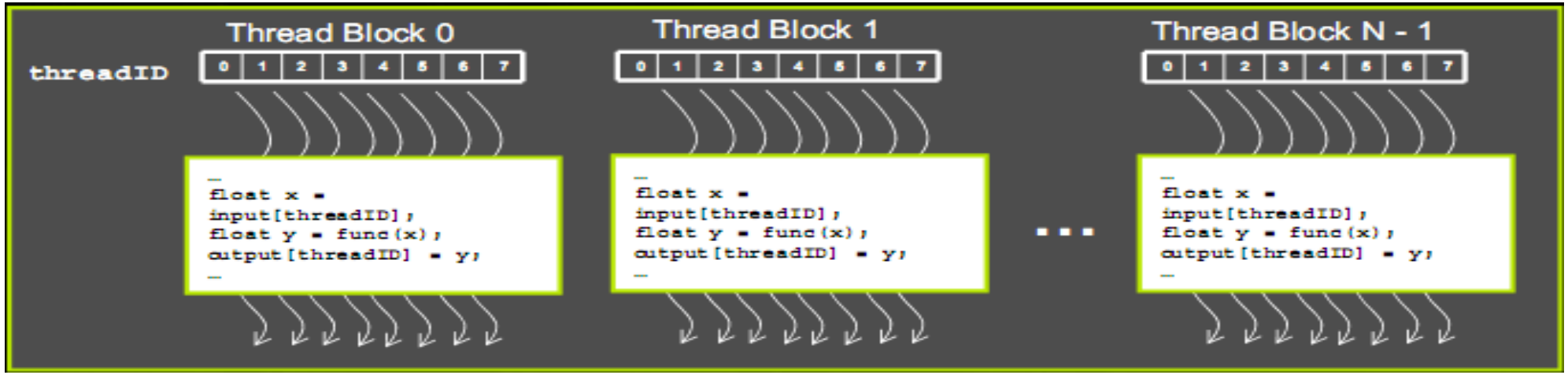


Fig.7. Hierarchy of parallel threads

- By definition, threads in the same block may synchronize with barriers

```
scratch[threadID] = begin[threadID];
__syncthreads();
int left = scratch[threadID - 1];
```

Threads wait at the barrier until all threads at the same block reach the barrier.

Transparent Scalability

- Threads blocks cannot synchronize
  - So they can run in any order, concurrently or sequentially
- This independence gives scalability:
  - A Kernel scales across any number of parallel cores

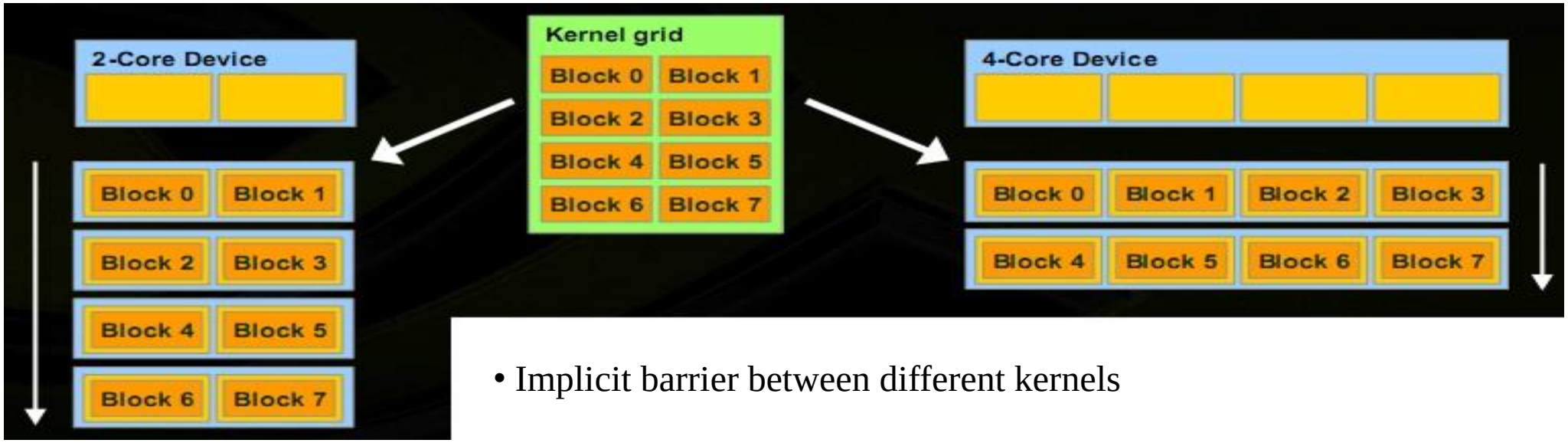


Fig.8. Kernel Grid with multiple core device

Programming Model: Memory (has Thread Hierarchy) Model

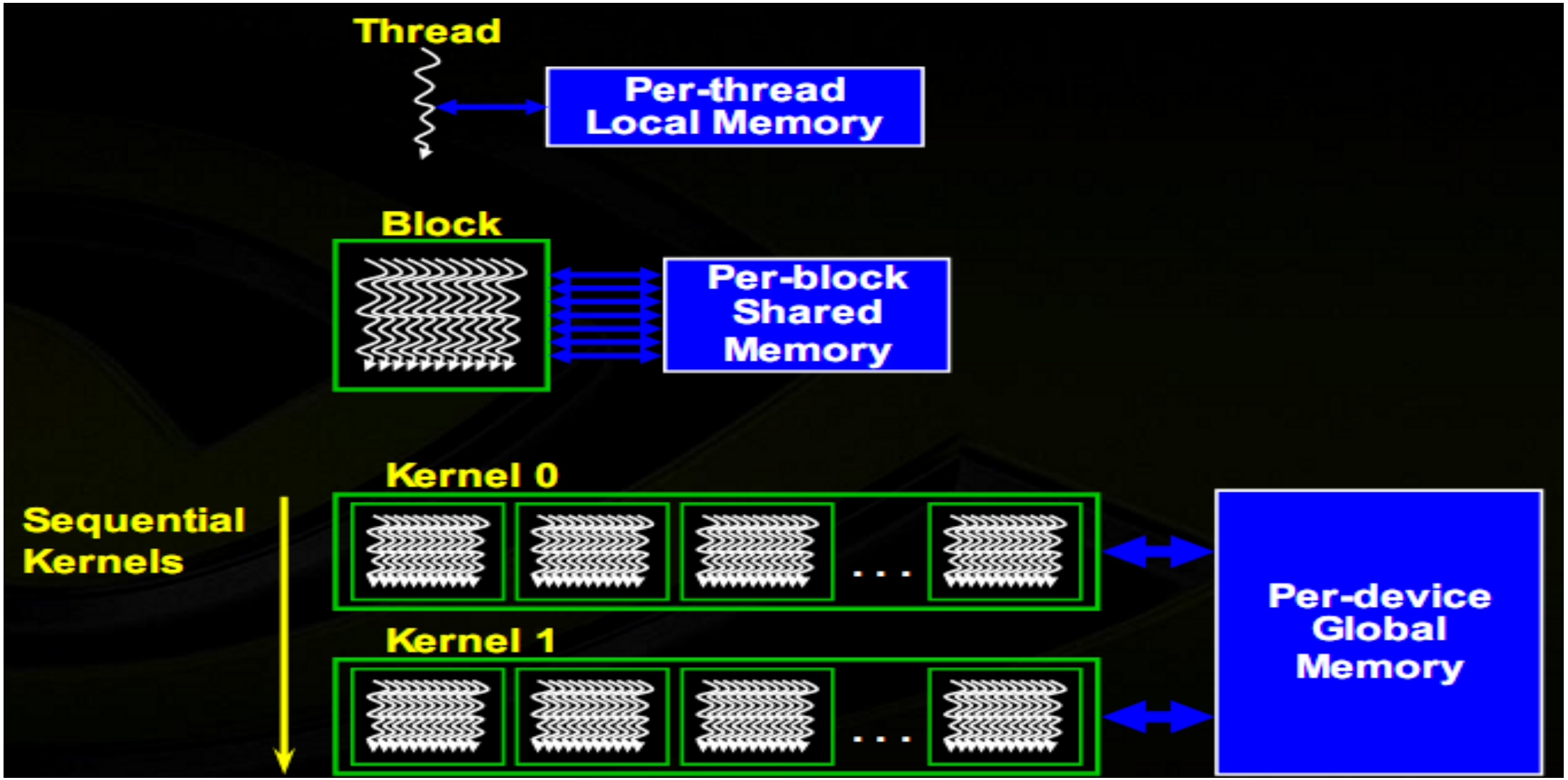


Fig.9. Thread processing model

Heterogeneous Memory Model

Fig.10. Memory model between host and devices

Memory Model : Computational Grid

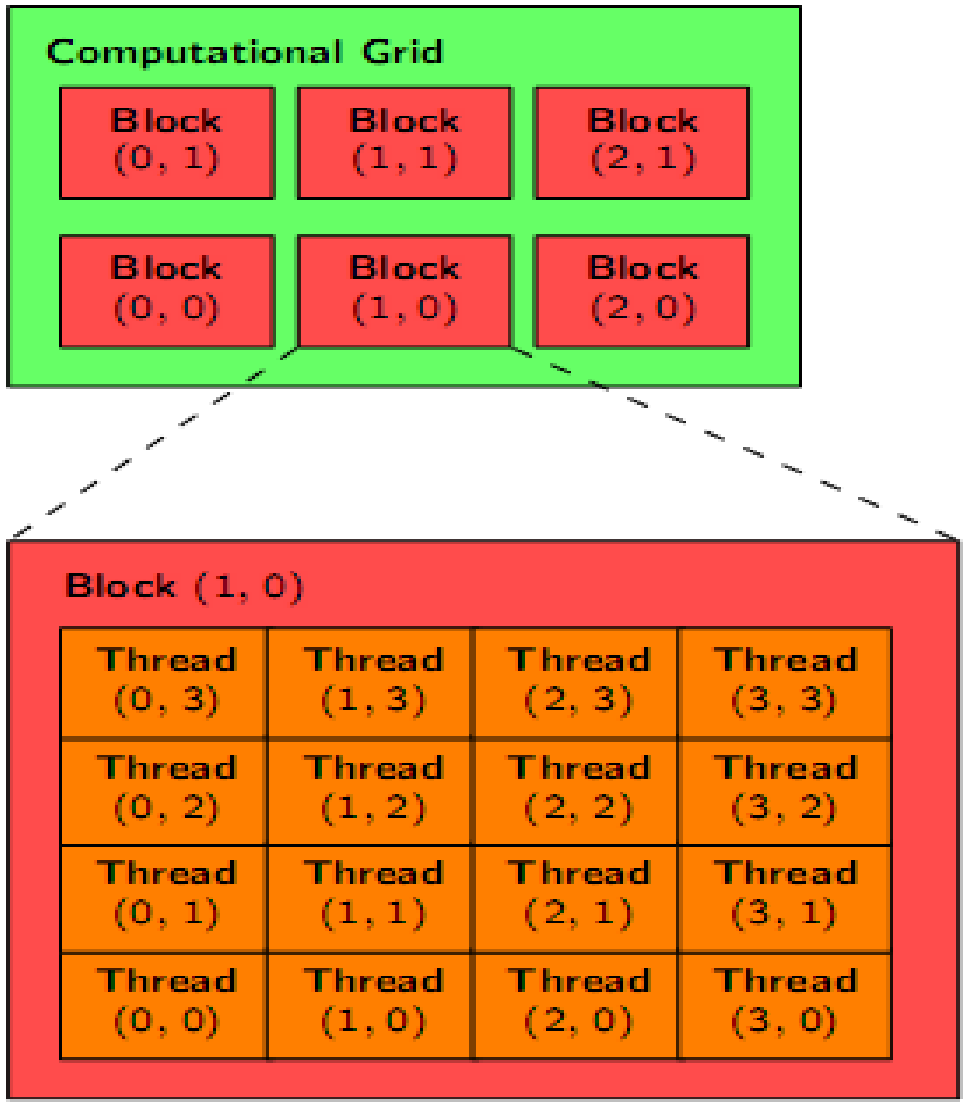


Fig.11. Basic need for Memory model

Memory Model

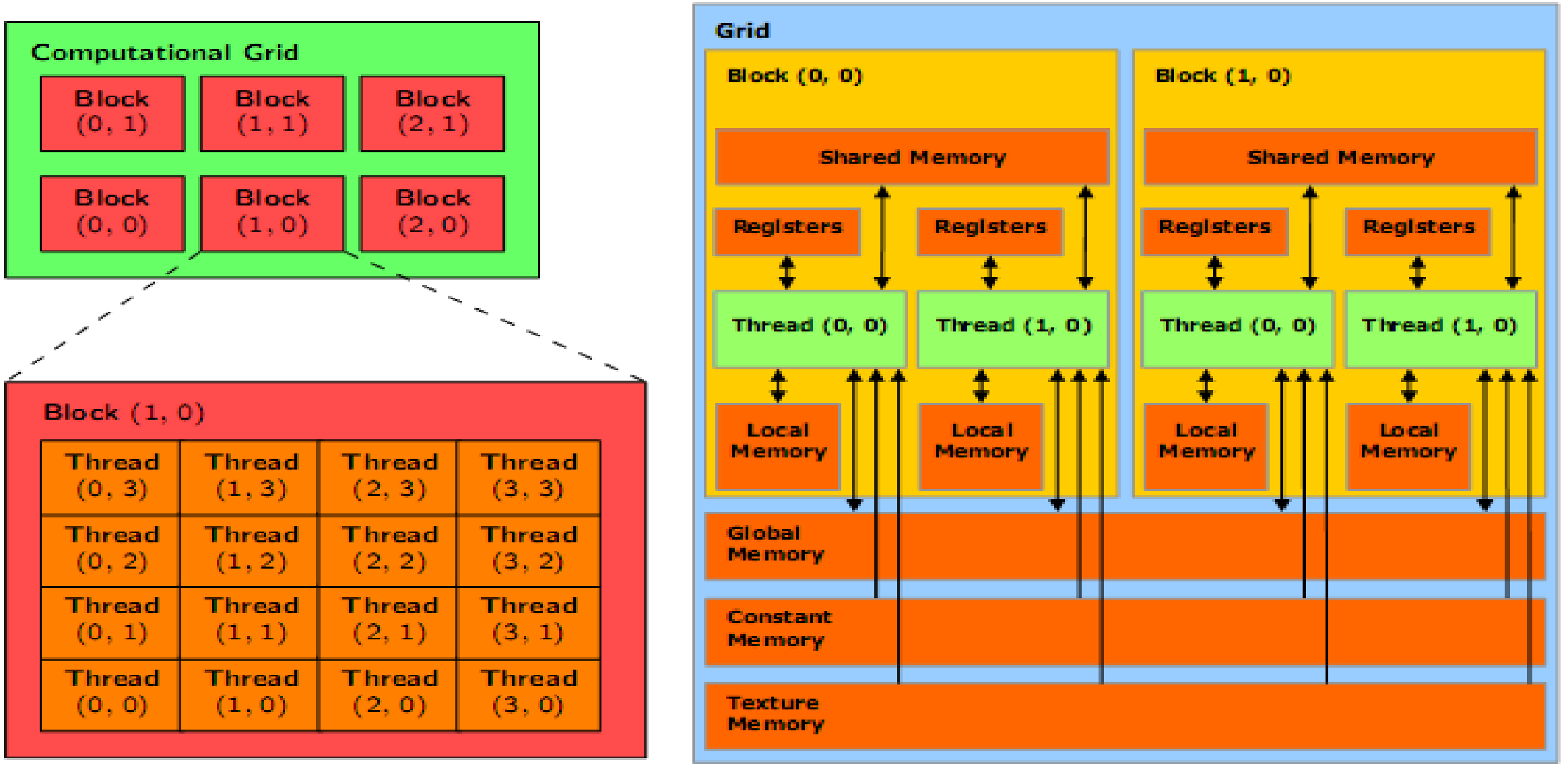


Fig.12. Interfacing of memories with per block

Hardware Implementation

- A set of SIMD Multiprocessors with On-Chip shared memory.

- Execution Model.

A set of SIMD Multiprocessors with On-Chip shared memory

- The device is implemented as a set of *multiprocessors*.
- Each multiprocessor has a Single Instruction, Multiple Data (SIMD) architecture.
- Each multiprocessor has on-chip memory of the four following types:
  - One set of local 32-bit registers per processor,
  - A parallel data cache or shared memory,
  - A read-only constant cache,
  - A read-only texture cache.

SIMD Multiprocessor

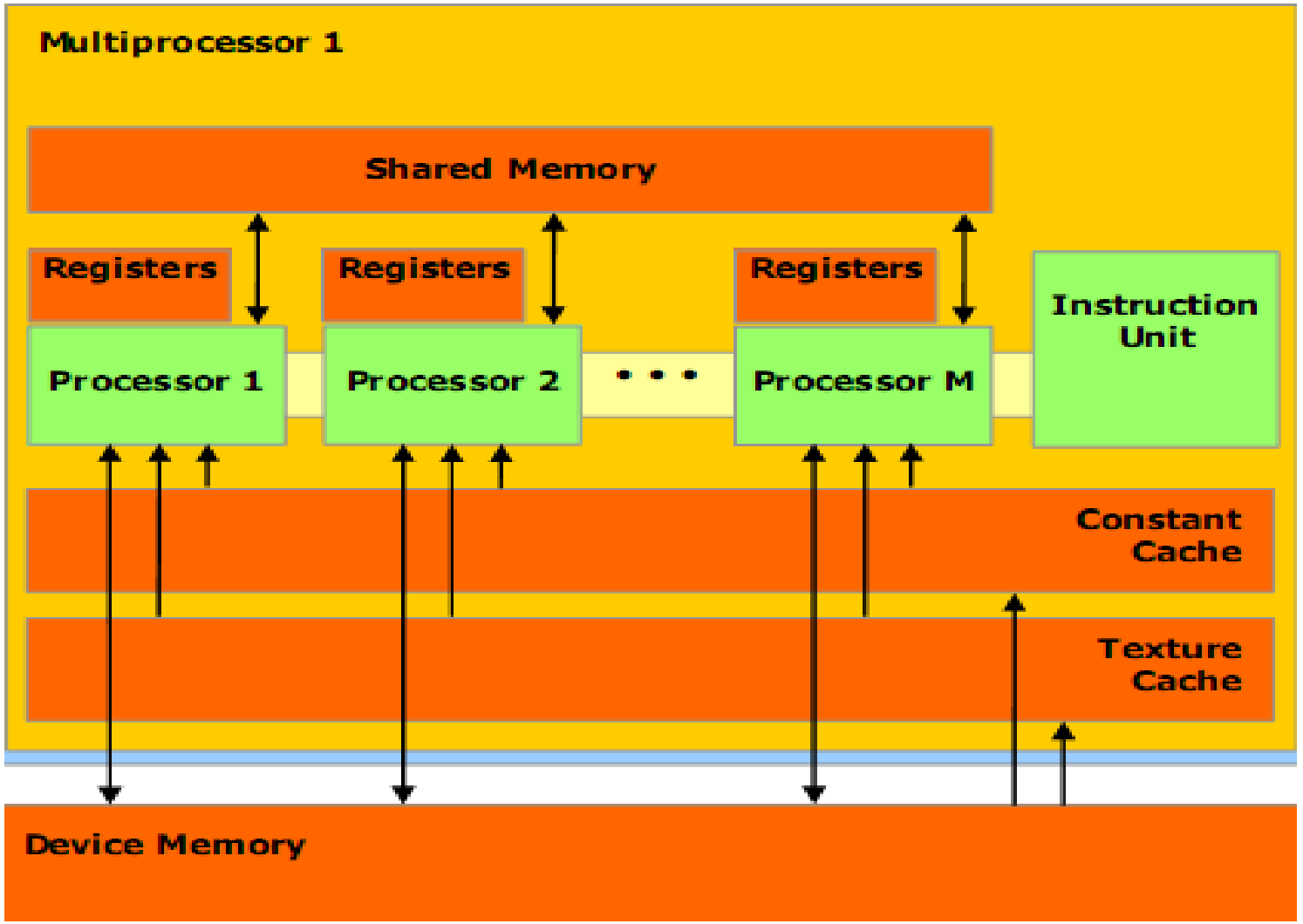


Fig.13. Single Instruction, Multiple Data processing

A set of SIMD multiprocessors with on-chip shared memory

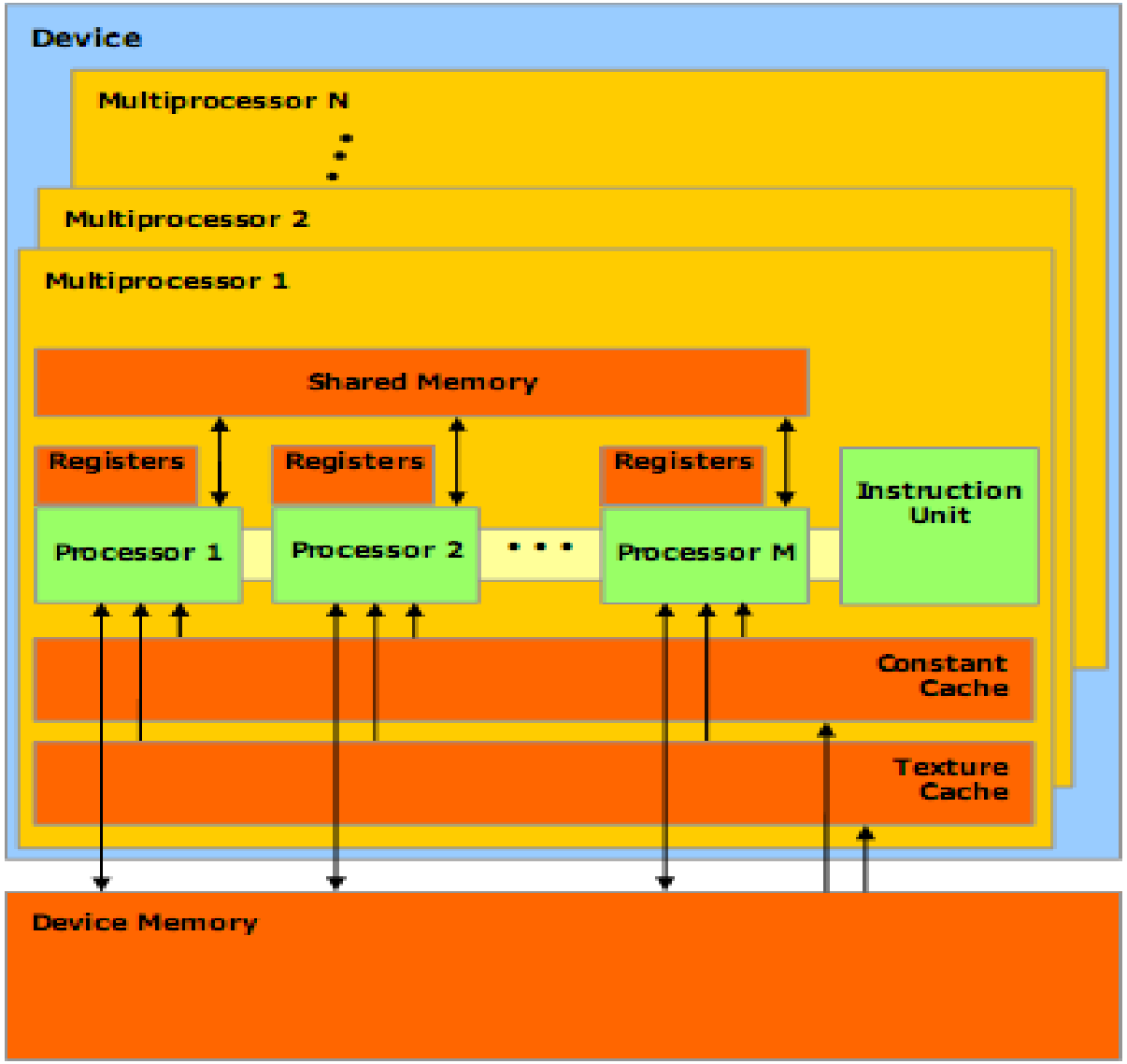


Fig.14. Set of SIMD multiprocessors

Execution Model

- Executed on the device by executing one or more blocks on each multiprocessor using time slicing.
- Each block is split into SIMD groups of threads called *warps*.
- Each of these warps contains the same number of threads, called the *warp size*.
- It is executed by the multiprocessor in a SIMD (Single Instruction Multiple Data) fashion.
- A *thread scheduler* periodically switches from one warp to another to maximize the use of the multiprocessor's computational resources.

CUDA Application Programming Interface.

- C Runtime for CUDA.
- Language Extension.
- Common Runtime Component.
- Device Runtime Component.
- Host Runtime Component.

Extension to the C programming language:

- GOAL: To provide a relatively simple path for users familiar with the C programming language.
- To easily write programs for execution by the device.

It consists of:

- A minimal set of extensions to the C language: allow the programmer to target portions of the source code for execution on the device.
- A runtime library split into:

➢ A host component

➢ A device component

➢ A common component

C Runtime for CUDA

- Handles kernel loading.
- Setting up kernel parameters.
- Launch configuration before the kernel launched.

C runtime for CUDA can perform:

- Implicit code initialization
- CUDA context management
- CUDA module management
- Kernel configuration
- Parameter passing

Language Extension

The extensions to the C programming language are four-fold:

- Function type qualifiers
- Variable type qualifiers
- A new directive
- Four built-in variables

Each source file containing these extensions must be compiled with the CUDA

compiler **nvcc**

Function type qualifiers

- **_device_**

  The _device_ qualifier declares a function that is:

  1. Executed on the device
  2. Callable from the device only

- **_global_**

  The _global_ qualifiers declares a function as being a kernel. Such function is:

  1. Executed on the device
  2. Callable from the host only

- **_host_**

  The _host- qualifier declares a function that is:

  1. Executed on the host
  2. Callable from the host only

Variable Type Qualifiers

- **_device_**

  The _device_ qualifiers declares a variable that resides on the device.

  1. Resides in global memory space.
  2. Has the lifetime of an application.
  3. Is accessible from the threads within the grid and from the host through the runtime library.

- **_constant_**

  The _constant_ qualifiers, optionally used together with _device_, declares a variable that:

  1. Resides in constant memory space
  2. Has the lifetime of an application
  3. Is accessible from all the threads within the grid and from the host through the runtime library.

- **_shared_**

  The _shared_ qualifiers, optionally used together with _device_

Common Runtime Component

The common runtime component can be used by both host and device functions:

Built-in Vector Types: These are vector types derived from the basic integer and floating-point types.

- The 1st, 2nd,3rd, and 4th components are accessible through the fields X, Y, Z, and W respectively.
- All come with a constructor function of the form **make_<type name>.**

  int2 make_int2(int x, int y);

- Which creates a vector of type int2 with value (x, y).

Device Runtime Component

- Mathematical Functions

- Synchronization Function: _syncthreads()

- Type Conversion Functions: The suffixed in the function below indicate IEEE-754 rounding modes:
  - Rn is round-to-nearest-even
  - Rz is round-towards-zero
  - Ru is round-up
  - Rd is round-down

- Texture Functions: tex1Dfetch()

Host Runtime Component

- Device management
- Context management
- Memory management
- Code module management
- Execution control
- Texture reference management
- Interoperability with OpenGL and Direct3D

Performance Guidelines

- Instruction Performance.
- Number of Threads per Block.
- Memory Optimization.
- Data Transfer between Host and Device.
- Benefits of Device, Shared, Local, Texture, and Constant Memory.

Programming Model: Managing Memory

- Host code manages device memory:
  - Allocate / free
  - Copy data
  - Applies to global and constant device memory (DRAM)
- Shared memory is statically allocated
- Host manages texture data:
  - Stored on GPU
  - Takes advantage of texture caching / filtering / clamping
- Host manages non-pageable CPU memory:
  - Allocate / free

Memory Model

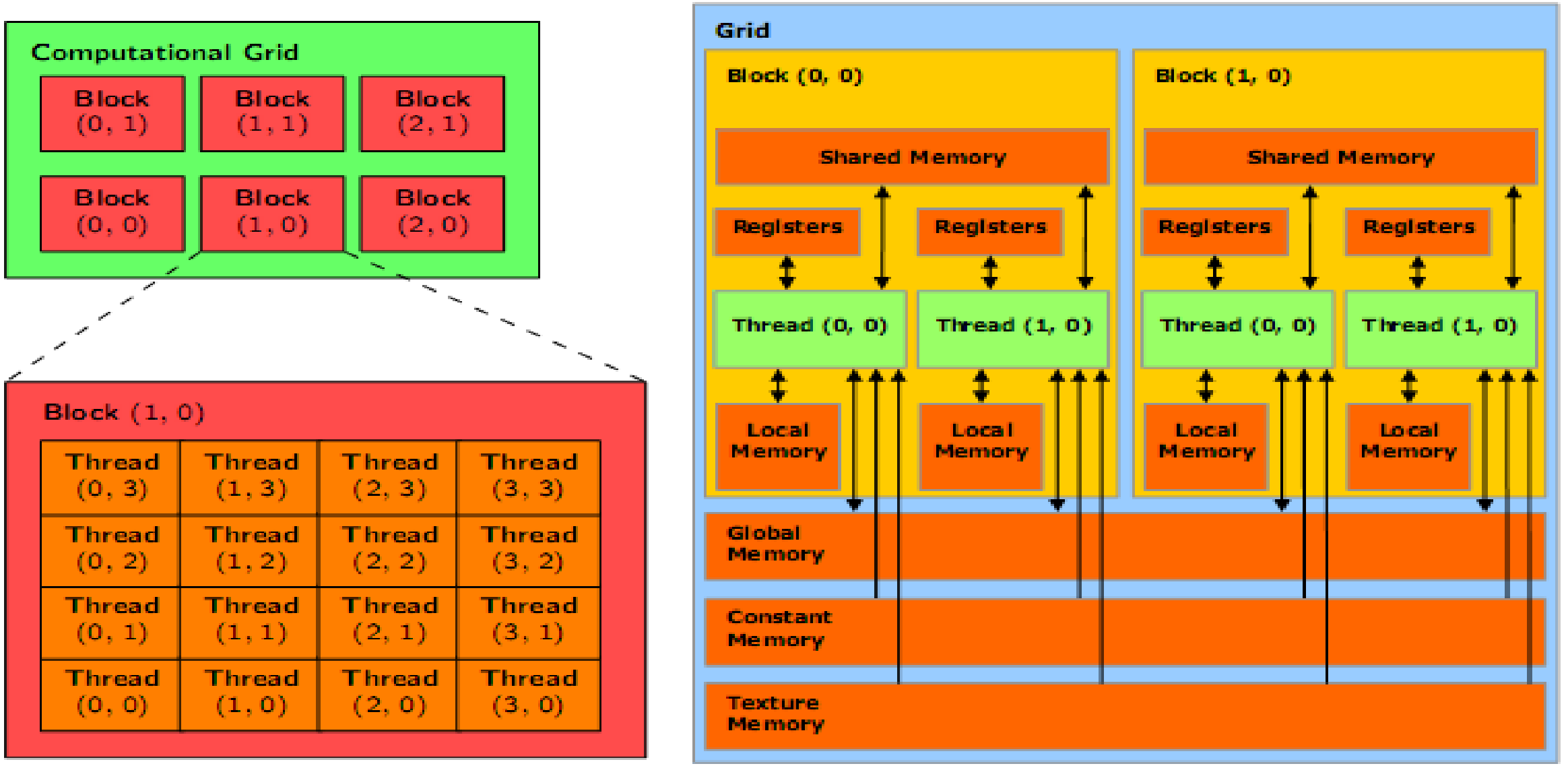
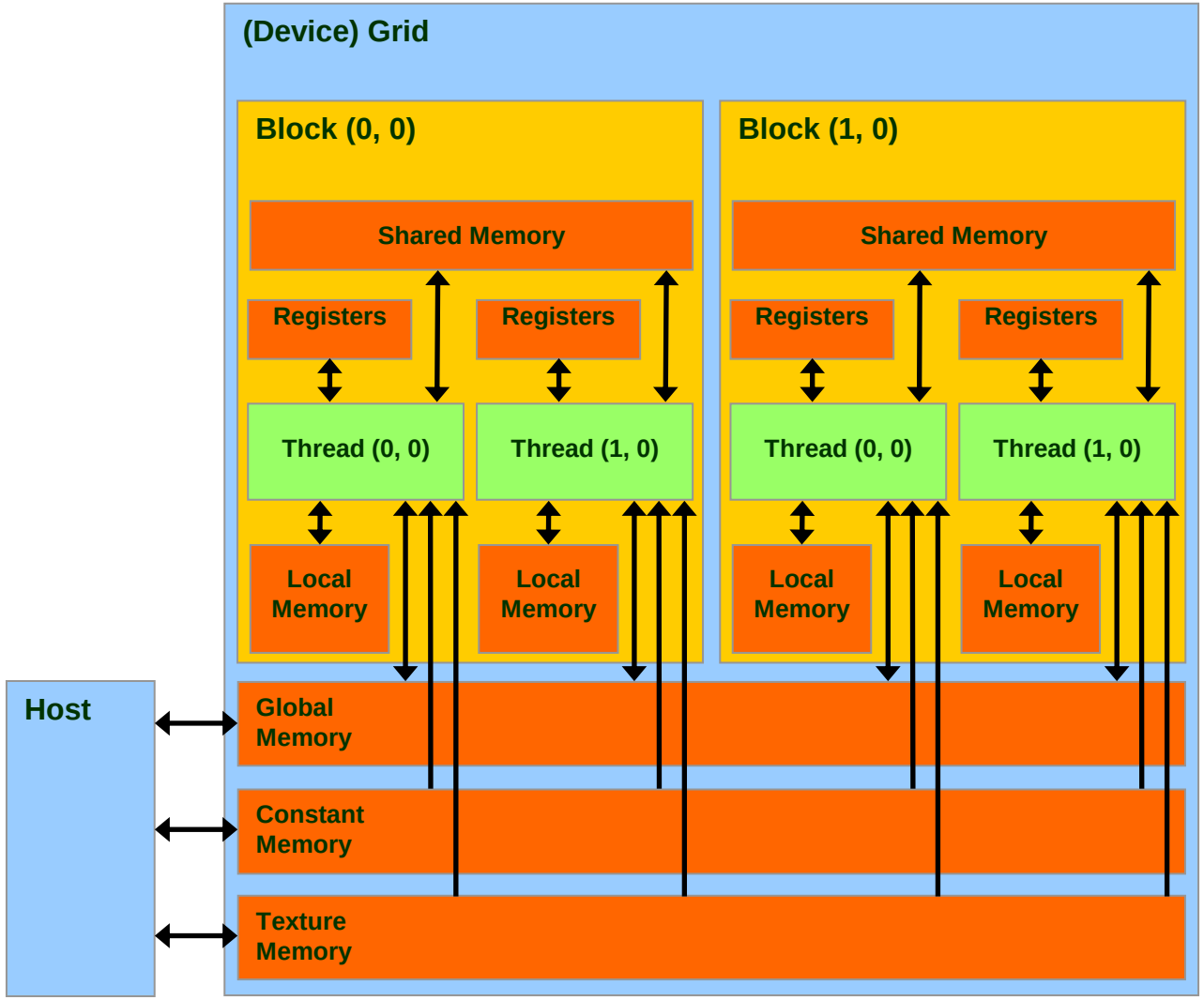


Fig.12. Interfacing of memories with per block

CUDA Device Memory Space Overview

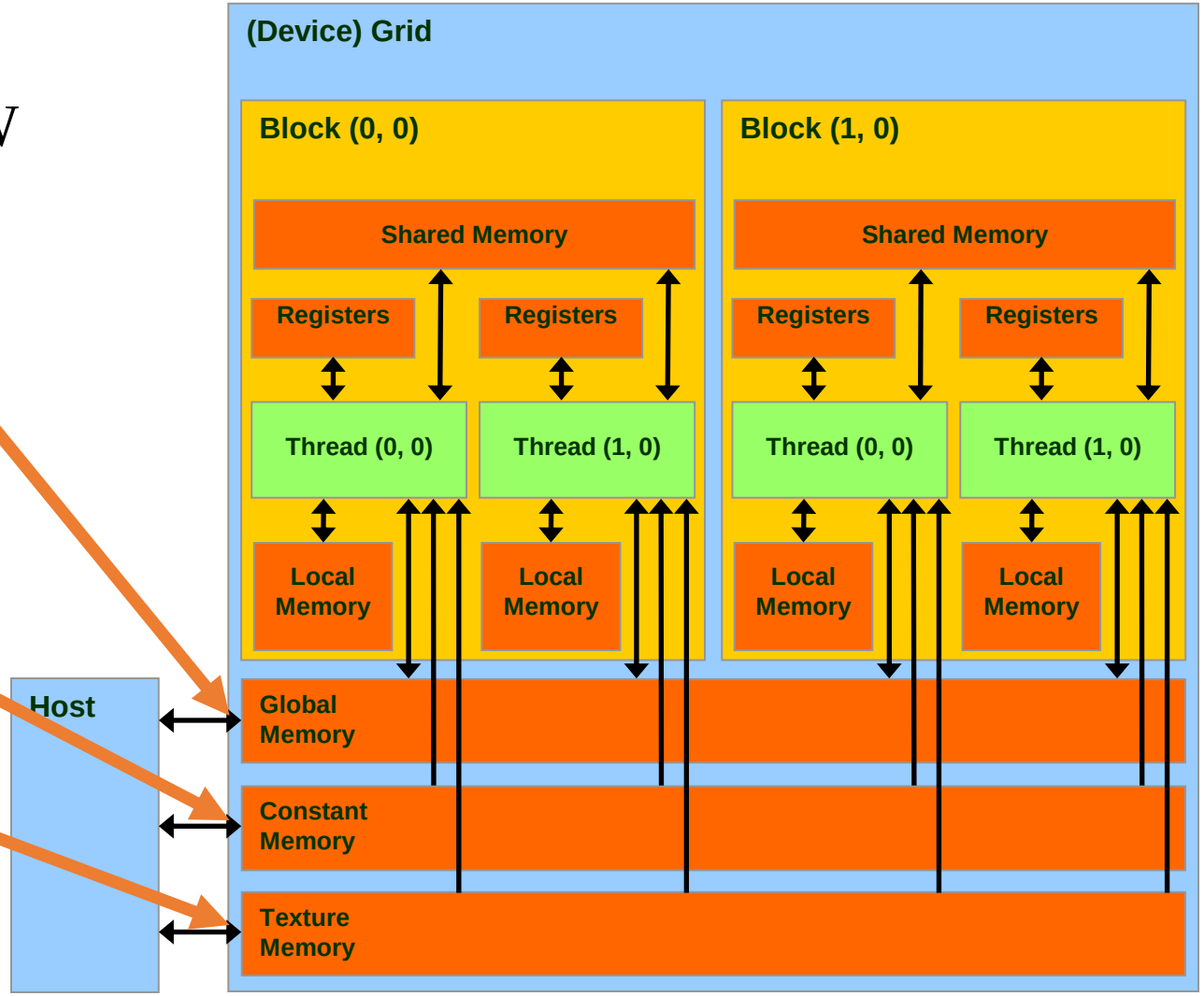
- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memories



Global, Constant, and Texture Memories (Long Latency Accesses)

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads



Courtesy: NDVIA

GPU Memory Allocation / Release

- cudaMalloc(void **pointer, size_t nbytes)
- cudaMemset(void *pointer, int value, size_t count)
- cudaFree(void* pointer)

```
int n =1024;
int nbytes = 1024*sizeof (int);
int *d_a = 0;
cudaMalloc((void**)&d_a, nbytes);
cudaMemset(d_a, 0, nbytes);
cudaFree(d_a);
```

Compliable Example

```
const int N = 1024;
const int blocksize = 16;

__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

- Store in source file (i.e. MatrixAdd.cu)
- Compile with nvcc MatrixAdd.cu
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

```
const int N = 1024;
const int blocksize = 16;
```

Set grid size

```
__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

- Store in source file (i.e. `MatrixAdd.cu`)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

```
const int N = 1024;
const int blocksize = 16;
```

Compute kernel

```
__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

- Store in source file (i.e. `MatrixAdd.cu`)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

```
const int N = 1024;
const int blocksize = 16;

__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

- Store in source file (i.e. `MatrixAdd.cu`)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

CPU Mem Allocation

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

```
const int N = 1024;
const int blocksize = 16;

__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

- Store in source file (i.e. MatrixAdd.cu)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f;
```

GPU Mem Allocation

```
  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

```
const int N = 1024;
const int blocksize = 16;

__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

- Store in source file (i.e. `MatrixAdd.cu`)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&
  cudaMalloc( (void**)&

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

Copy data to GPU

```
const int N = 1024;
const int blocksize = 16;

__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

- Store in source file (i.e. `MatrixAdd.cu`)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cu
  cudaMemcpy( bd, b, size, cu
```

Execute kernel

```
  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

```
const int N = 1024;
const int blocksize = 16;

__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

- Store in source file (i.e. `MatrixAdd.cu`)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<
```

Copy result back to CPU

```
  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

```
const int N = 1024;
const int blocksize = 16;

__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

- Store in source file (i.e. MatrixAdd.cu)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd,
```

Clean up and return

```
  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

Element wise Matrix Addition

CPU Program

```
void add_matrix
  ( float* a, float* b, float* c, int N ) {
  int index;
  for ( int i = 0; i < N; ++i )
    for ( int j = 0; j < N; ++j ) {
      index = i + j*N;
      c[index] = a[index] + b[index];
    }
}

int main() {
  add_matrix( a, b, c, N );
}
```

CUDA Program

```
__global__ add_matrix
  ( float* a, float* b, float* c, int N ) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}

int main() {
  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );
}
```

CPU Program

```
void add_matrix
  ( float* a, float* b, float* c, int N ) {
  int index;
  for ( int i = 0; i < N; ++i )
    for ( int j = 0; j < N; ++j ) {
      index = i + j*N;
      c[index] = a[index] + b[index];
    }
}

int main() {
  add_matrix( a, b, c, N );
}
```

CUDA Program

```
__global__ add_matrix
  ( float* a, float* b, float* c, int N ) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}

int main() {
  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );
}
```

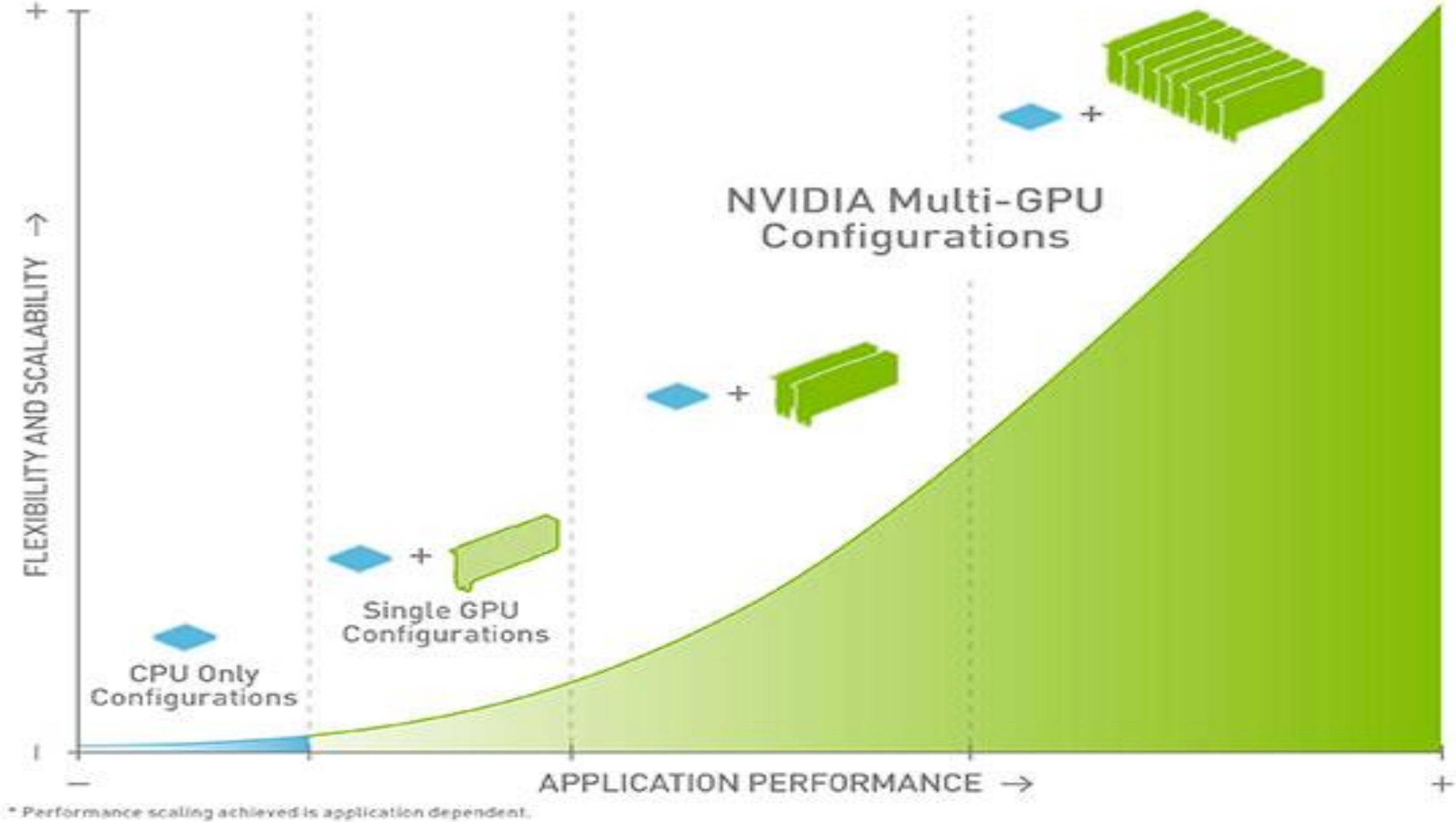
The nested `for`-loops are replaced with an implicit grid

Introduction of Multi-GPU

- NVIDIA® Maximus® technology transformed the design process by combining the industry-leading graphics capability of NVIDIA Quadro® graphics processing units (GPUs) and the high performance computing power of NVIDIA Tesla® GPUs.

  - **Substantial time savings**—A multi-GPU system helps address the pressures of delivering a high-quality product to market more quickly by providing ultra-fast processing of your computations, renderings, and other computational and visually intensive projects.

  - **Multiple iterations**—The ability to revise your product multiple times in a resource and time-constrained environment will lead to a better end result. Completing each iteration of your automobile, animated movie, or seismic data processing faster, leads to additional refinements.



* Performance scaling achieved is application dependent.